Contents

[1 Analysis 2](#_Toc3329884)

[1.1 The Problem 2](#_Toc3329885)

[1.1.1 Existing Solution 2](#_Toc3329886)

[1.1.2 Student Robotics Simulator 3](#_Toc3329887)

[1.1.3 Further Client Discussions 3](#_Toc3329888)

[1.2 Prototyping and Research into Techniques 4](#_Toc3329889)

[1.3 Requirements 6](#_Toc3329890)

[1.4 Modelling and Object Diagrams 9](#_Toc3329891)

[1.4.1 Objects for Physics 9](#_Toc3329892)

[1.4.2 Objects for Scheduling 9](#_Toc3329893)

[1.4.3 Modelling Vision 11](#_Toc3329894)

[2 Documented Design 12](#_Toc3329895)

[2.1 Source Files and Module Organisation 13](#_Toc3329896)

[2.2 Object Diagrams 15](#_Toc3329897)

[2.3 Data 16](#_Toc3329898)

[2.3.1 Global Variables 16](#_Toc3329899)

[2.3.2 Configuration Files 17](#_Toc3329900)

[2.4 Algorithms 17](#_Toc3329901)

[2.4.1 Multi-Threading and Processing 17](#_Toc3329902)

[2.4.2 Vision 23](#_Toc3329903)

[2.5 User Interface 28](#_Toc3329904)

[2.5.1 Command Line Syntax 28](#_Toc3329905)

[2.5.2 SimDisplay 28](#_Toc3329906)

[3 Testing 29](#_Toc3329907)

[3.1 Flow Control Testing 29](#_Toc3329908)

[3.1.1 Verifying the Startup Sequence: 29](#_Toc3329909)

[3.1.2 Verifying the Main Loop Sequence: 30](#_Toc3329910)

[3.1.3 Verifying the Shutdown Sequence: 30](#_Toc3329911)

[3.2 Vision Testing 31](#_Toc3329912)

[3.2.1 vector3 Unit Test 31](#_Toc3329913)

[3.2.2 SimVision Component Test 32](#_Toc3329914)

[3.3 Testing Other Requirements 35](#_Toc3329915)

[3.3.1 Sample of Test Results 36](#_Toc3329916)

[3.4 System Testing 36](#_Toc3329917)

[3.5 End User Level Testing 37](#_Toc3329918)

[4 Evaluation 38](#_Toc3329919)

[4.1 Requirements Review 38](#_Toc3329920)

[4.2 Known Issues 42](#_Toc3329921)

[4.2.1 Client Programs Not Returning 42](#_Toc3329922)

[4.2.2 Orientations of Markers Incorrectly Returning 42](#_Toc3329923)

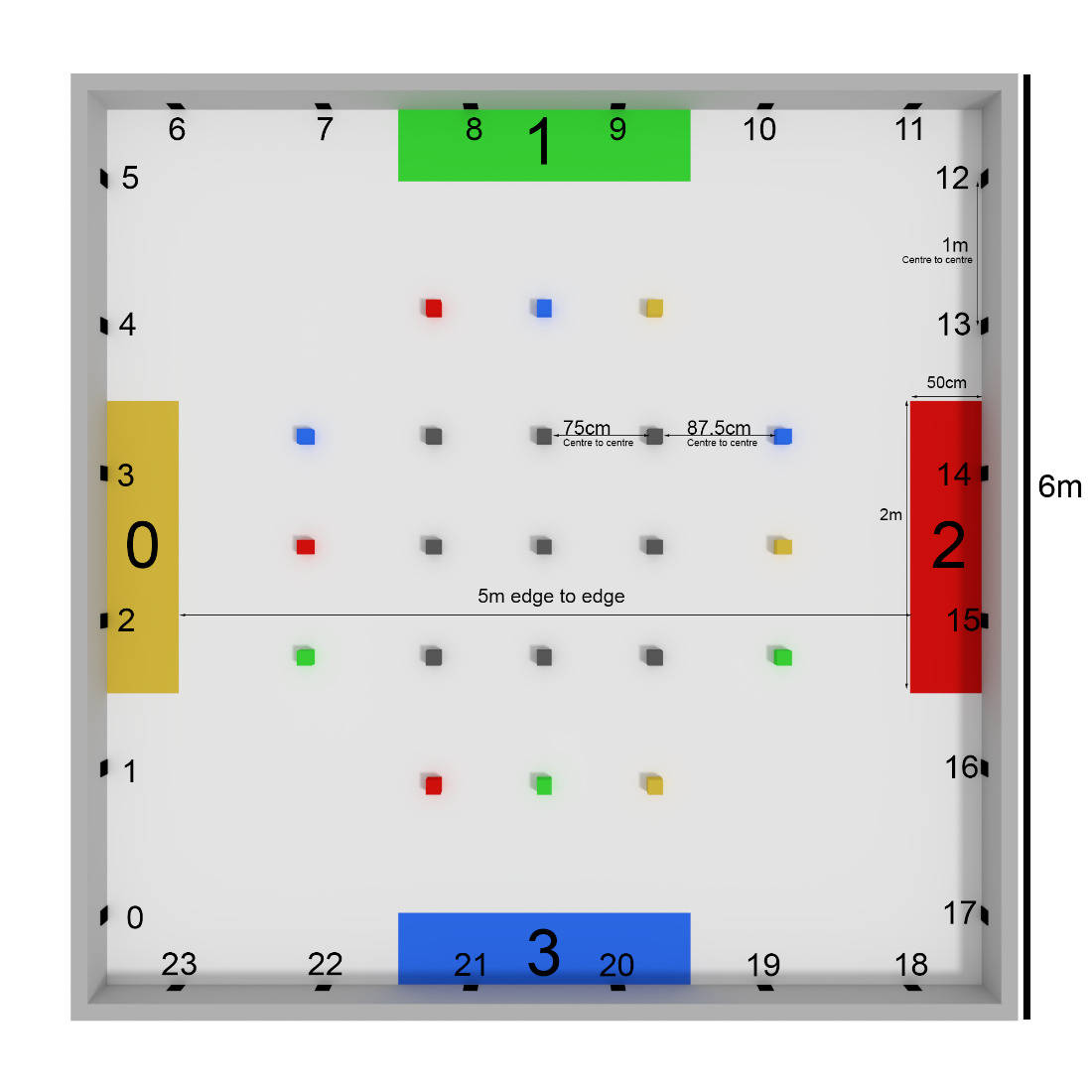
[4.3 Client Feedback 42](#_Toc3329924)

# **Analysis**

## The Problem

RoboCon is a robotics competition for students in Years 9 through 11. They must build and program a robot over the course of a few months to participate in a competition fully autonomously.

This year’s competition, Gold Rush, takes place in a 6m by 6m arena, with 25cm vision markers placed on the walls, 5cm above the floor, placed every metre starting from 0.5m in. Additionally, each team has a coloured area called their “zone” which their robot must start in.  
The arena contains 21 cubic tokens, 11cm in all dimensions, with a 10cm square marker in the middle on all sides. They are arranged in predefined positions. The marker for each object is unique and can be used to identify the part of the wall or the type of token.



My client is David Massey, organiser of RoboCon, has requested that I create a robot simulator like Student Robotics’ simulator (<https://www.studentrobotics.org/docs/programming/simulator>). The end users would be the teams of teenage students, early in the development process, who want to test algorithms before they had built a robot, or test while robots were in use elsewhere.

### Existing Solution

Currently, teams must build a working prototype of their robot of their robot available before they can begin writing algorithms. To perform a full test, they would need to construct an arena and three additional robots, and the only debugging tools available to them are print statements and the raw camera output.

The competition is derived from a national competition called Student Robotics, and they provide a simple robotics simulator. This simulates the motors, camera and a simple “grab” mechanism on a predefined robot. This means teams can start working on algorithms immediately, and more than one person can begin testing things at the same time.

Unfortunately, our robot hardware and API are slightly different to the SR hardware and is therefore not accurately simulated by their simulator. Additionally, some rules and specifications like the Arena size or scoring conditions also different.

### Student Robotics Simulator



The original request was to make something “like the Student Robotics simulator”, which can be found at <https://www.studentrobotics.org/docs/programming/simulator>. The simulator allows up to four robots with different test programs to be simulated at once, and uses a very similar API to the base Student Robotics one – with a handful of differences:

* The “Marker” object returned by the See() function is slightly different in format to the one returned by a real robot, and contains slightly less information.
* Servos are not supported.
* As a substitute for servos, the robot is equipped with a “grabber”, which can grab a token that’s directly in front of the robot. This is accessed with “R.grab()” and “R.release() functions.

### Further Client Discussions

As a basis for discussion, I drafted an initial draft set of features and use cases, based off the SR simulator:

* It must be possible to use the same program file in the simulator as the robot.
* It must be possible to watch a graphical display of the simulated arena in real time.
* The robot must be able to use Motor() commands to drive around the simulated environment.
* The see commands must return all markers “visible” to the simulated robot. Distance, obstructions, motion blur and noise may stop a marker from being visible.
* When an object collides with another object, the physics must be realistically modelled.
* It should be possible to simulate multiple robots in the arena, using different programs.
* Noise and other robot parameters (such as robot size and shape) should be easily changed using a configuration file.
* It should be possible to run repeated simulations unattended and access a log of the results at the end.

The first six requirements described the capabilities of the SR simulator, and the last two were extensions that he had mentioned in the initial request. Mr Massey said that all of these features would be ideal, and also came up with some additional requirements:

* The base speed of the motors should be configurable using the configuration file, to simulate different types of motor and wheel.
* All simulated operations must be logged with timestamps, and it should be possible to use these logs to “replay” a simulation.
* There must be toggleable “vision beams” on the graphical display when See() is called, which show the field of view and obstructions.

We also discussed a number of features that I was expecting to only implement in a limited form, to avoid time, complexity or engine restrictions. We agreed that all of these features would be acceptable limitations.

* All physics components of the simulation will be calculated in 2D.
* All robots must be rectangular/cuboid.
* Simulated robots cannot have servos (and hence articulated limbs), and the direction of the camera is fixed (looking out of the front of the robot).
* Wheel slip and other dynamic aspects of an accelerating system will be highly simplified if simulated at all.

There was one exception – while I had originally planned on simplifying the vision to 2D, Mr Massey requested that I did it in 3D, as in real competitions, many robots have high mounted cameras and the vertical positioning has a great impact on the vision of the robot.

## Prototyping and Research into Techniques

In order to model collisions and motion for the blocks and robots, I looked into physics simulation for blocks. My findings for 2D polygons were as follows:

* With a single entity, calculating the change in position and orientation with a given force is easy.
* It is also fairly manageable when calculating collisions with an immovable wall, or a second movable entity with no forces.
* With more than two entities in consideration, or with arbitrarily complex systems, it becomes much harder, and quickly far surpasses anything I can do with A-Level knowledge. A useful set of articles for a problem similar to mine can be found at <http://chrishecker.com/Rigid_Body_Dynamics>.
* In addition to complex vectors and mechanics, complications such as friction being a velocity dependant force made it impractical to write my own physics solver.

In researching this, most sources recommended that you use an existing physics library. I found the following:

* PyBullet (<https://pypi.org/project/pybullet/>) and PyODE (<http://pyode.sourceforge.net/>) are popular physics solvers for python, though they work in 3D, making them excessive for my requirements, and have “Getting Started” PDFs which are 70 pages long.
* PyBox2D (<https://github.com/pybox2d/pybox2d>) is a solver far more suitable for my purposes – it works in 2D and has a convenient “debug draw” option, which means I can test the solver without needing to write a rendering engine first. Unfortunately, it was last updated three years ago and may not work with Python 3.
* PyMunk (<http://www.pymunk.org/en/latest/>) is derived from PyBox2D and is actively being supported. Prototyping indicates that this engine works for my purposes and is ideal for the project.

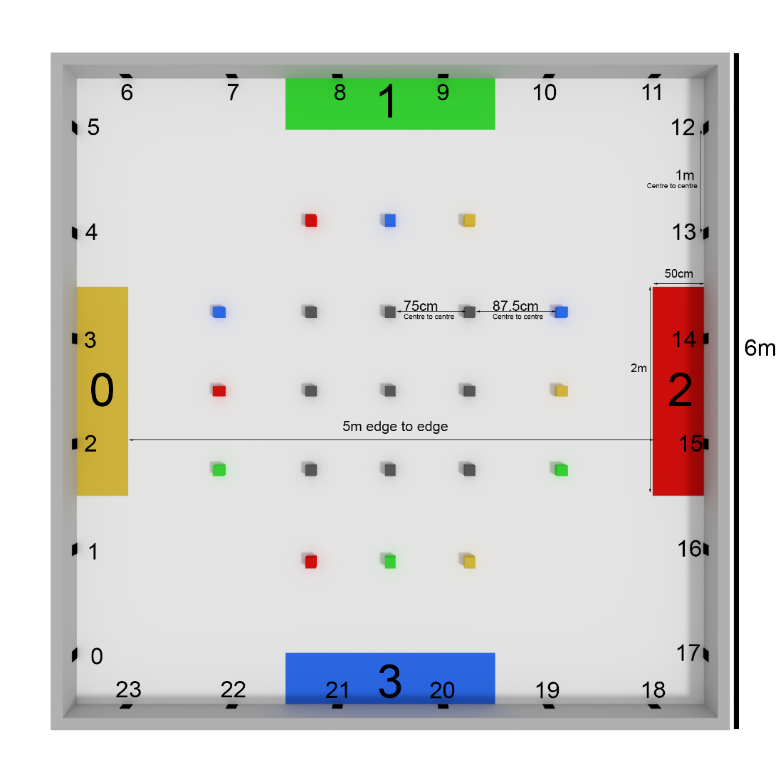
Research quickly indicated that if I wanted to execute python programs, I would not be able to interpret them myself (as that is exceedingly complicated) and I would need to instead run the programs in a real interpreter and intercept calls to the robot. In order to send and receive calls between programs, I decided to use XML-RPC (<https://docs.python.org/2/library/xmlrpclib.html>) to send and receive messages between a client and server machine. Through prototyping, I was able to show that it was possible to use it on a school machine.

My research into detecting if an object is obstructed by another object indicated that I could do it using my existing knowledge from my Further Maths A Level. During prototyping, I used a guide to write a simpler algorithm for 2D space that identified if a bounded line was obstructing another line (<https://ncase.me/sight-and-light/>) – this used many of the ideas that were then applied to the 3D simulation.

## Requirements

Requirements are categorised by topics, which are ordered approximately by their importance. Stretch requirements are denoted with an [Ext].

1. I will build a simple robot simulator, capable of being given up to four python programs for a robot and running them in a virtual environment to mimic the Hills Road Robocon competition.
2. Simulations must be orchestrated in order to allow for:
   1. Multiple robot programs must be able to connect to the same simulated arena. Programs must be started together and stopped after a predetermined time expires.
   2. When the simulation time expires, the scores of the competition must be accessible. They are calculated as follows:
      1. Every token will score points for a team, depending on if it is in a zone or touching a robot.
         1. “Ore” tokens score 1 point for the robot’s team if touched, and 5 points for the zone’s team if in a zone.
         2. “Gold” tokens each have a team associated and score 3 points if touched by their corresponding team’s robot, 7 points if in their corresponding team’s zone, -1 point if touched by a different team’s robot, and -2 points if in a different team’s zone.
      2. Tokens can only score once, and will score for the highest absolute value they are valid for, so -2 beats +1. If more than one robot is touching a token, neither can score for it.
      3. Finally, robots earn a single point for their team if at any point in the competition they fully leave their zone.
   3. Multiple robots must be able to use different programs and different configuration settings.
   4. [Ext] The entire process must be repeatable for automated testing.
3. It must be possible to view the simulation taking place:
   1. Must display simulation in real time –the positions and orientations of all objects must be updated as they change in the simulation.
   2. Objects associated with a team (gold, zones, robots) must be coloured according to the team they belong to.
   3. When the vision function is called, a triangle representing the field of view of the robot must be drawn. Visible objects will be highlighted.
   4. Must display the standard output of the robots in real time – all print statements must be displayed to the user as they’re sent.
4. The simulator must work with unmodified code, though a small number of exceptions are allowable:
   1. Robot servos and GPIO collections will exist like in an actual robot, but will be empty, so programs that attempt to use them will not work.
   2. A small number of functions (such as *sleep* or *print*) will need to be changed. These changes must be possible using only simple Find and Replace tools.
      1. [Ext] The syntax changes are made automatically by the simulator, so the simulator can input completely unmodified code.
5. The simulated arena must be set up as per the rules of the competition:
   1. Immovable and impassable walls must constrain the arena to a 6x6 meter square.
      1. They have 24 markers positioned 0.5m above the ground – each wall has 6 markers, with their centres spaced 1m apart, starting 0.5m from the corner.
      2. The markers each have an Id number, which begins at 0 at the bottom of the leftmost wall, and increases clockwise.
   2. Zones of dimensions 0.5m by 2m are positioned against the centre of each wall, and each correspond to a team.
   3. 21 tokens populate the Arena:
      1. Tokens are 0.11m cubes, with a 0.1m square marker centred on each face. This marker displays the Id of the token.
      2. “Ore” tokens have unique Ids ranging from 32 to 41. Each team has their own “Gold” tokens, which have unique Ids ranging from 42 to 44 for team 0, 45 to 47 for team 1, etc.
      3. “Gold” tokens must begin the game placed 1.375 meters from the arena walls – each team has one “Gold” token in front of every other team’s zone, positioned as in the diagram. 0.875m inwards from that, the “Ore” tokens must be placed 0.75m apart from each other.
      4. [Ext] The positions of the tokens can be configured using a configuration file, to improve forwards compatibility with future competitions.

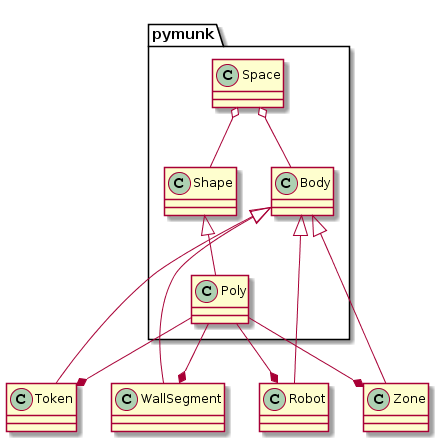


1. 2D physics should be calculated in the arena to simulate an actual competition. This is done almost entirely using pymunk’s inbuilt functions, though there are some specifications that I need to implement:
   1. Different objects in the arena have different masses:
      1. Walls are immovable.
      2. Tokens are (almost) weightless compared to robots.
      3. Robots can have varying weights.
   2. If an object is not undergoing any forces, it should stop almost immediately, to simulate the high friction carpet.
2. Changing the motors property of the robot in the test program must drive the robot in the simulation.
   1. The robot has two motors, with powers ranging from -100 to 100.
   2. Every physics step, the robot will move forwards (or backwards) depending on the powers set in the test program.
   3. The simulation should apply the correct rotational moment if the two motors have a different power.
3. Calling the see() function in the test program must return a list of markers visible to the robot in the simulation.
   1. A marker is visible if:
      1. All four corners are entirely unobstructed.
      2. All four corners are within the field of view.
      3. The sides of the resultant parallelogram projection onto the camera are longer than the configured threshold for the viewing robot.
   2. Markers must be returned in the format specified in the documentation (<https://hr-robocon.org/docs/vision.html>). In addition to following the object structure, I will need to return:
      1. Information about the marker, including:
         1. The Id of the marker.
         2. The type of the marker (wall or token).
         3. The type of token (none, ore, gold, or fools gold). This is from the perspective of the robot, so if the robot is on team 2, team 2’s gold is returned as gold.
         4. The size of the marker in metres.
         5. The resolution of the image that the marker was taken in.
         6. The time at which the image was taken.
      2. For every corner of the marker, and its centre, I need to return:
         1. Its position in a Cartesian coordinate system relative to the camera.
         2. Its position in a Polar coordinate system relative to the camera.
         3. The position in the raw image that the camera took.
      3. The orientation of the marker (relative to the camera) must be returned.
   3. If an object is moving, any markers on it cannot be seen (though it can still obstruct other markers). If a camera is moving, it will return no markers. These restrictions do not apply if motion blur is disabled for that robot in its configuration options.
4. The user must be able to change a configuration file (per team) to customise several parameters regarding that team’s robot:
   1. Starting position of robot (relative to their zone).
   2. Width, Length and Height of robot.
   3. Mass of robot.
   4. Distance between the wheels (axle length) of robot.
   5. Height of camera above the ground.
   6. Base speed of motors.
   7. The Field of View of the camera.
   8. The extent of simulated “hardware noise”:
      1. Motor offset range (maximum possible difference between the motors).
      2. Camera noise – necessary pixel threshold, with random variance factor.
   9. Toggling motion blur.
5. [Ext] It should be possible to save and replay a simulation:
   1. At every timestep, enough information about the simulation must be recorded to allow to be “reconstructed” for the display:
      1. The positions and orientations of all movable objects.
      2. The times when a robot called print(), and what it output.
      3. The times when a robot called its vision function, and the objects it saw.
   2. Must be able to playback a competition from a file.
   3. Must be able to pause, speed up, slow down or jump in the “timeline” of the playback.

## Modelling and Object Diagrams

### Objects for Physics

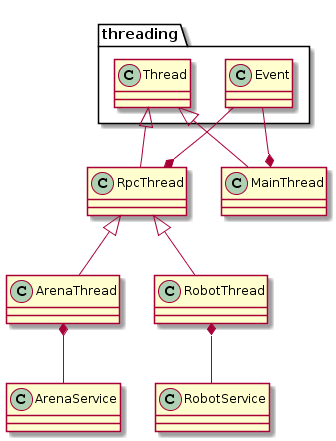
In order to use pymunk as my physics simulator, I need to represent the items in the arena as objects of classes derived from pymunk’s Body class. Additionally, a body needs one or more Shapes (defining its dimensions and mass). In my simulation all bodies have a single box shape, which is constructed with pymunk’s Poly class.



### Objects for Scheduling

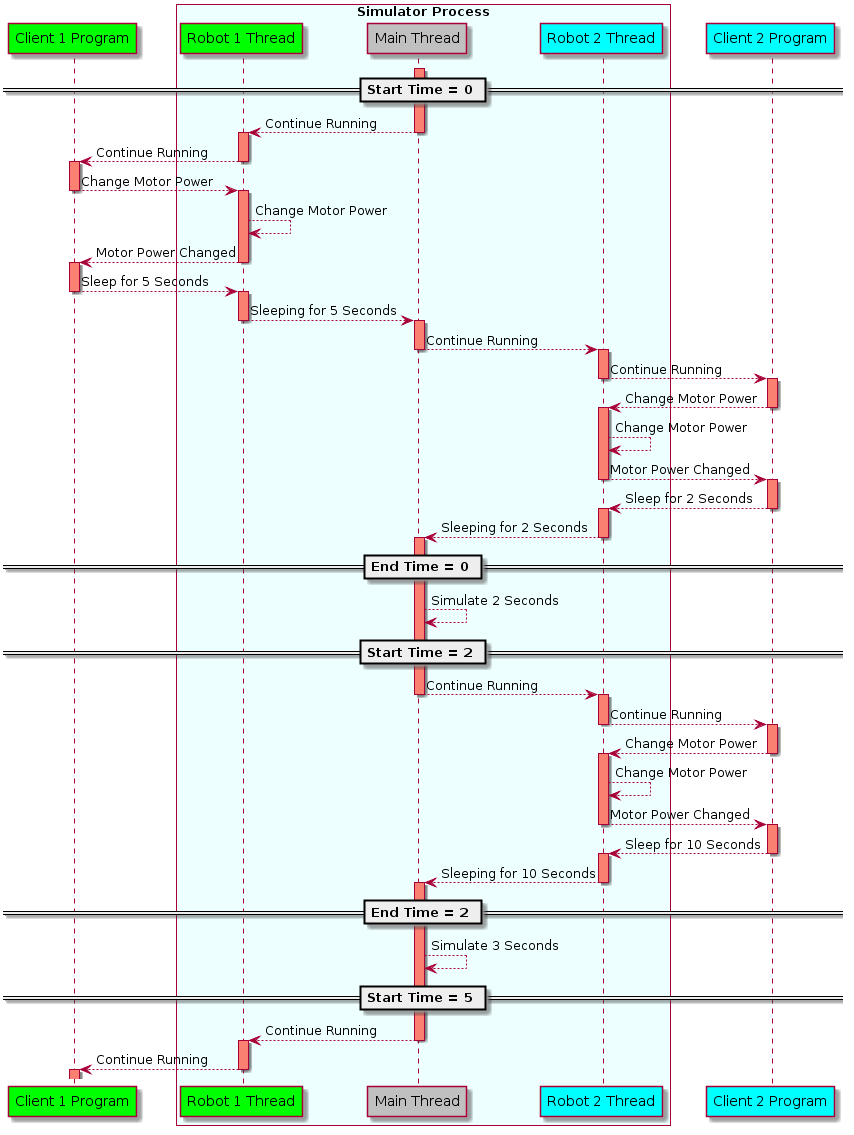
In order to execute a client program to test, I am going to run it as a sub-process (so the job of interpreting python is handled by the python interpreter), and intercept calls made to the robot library by replacing it with my own custom library with the same API. This approach also means that I need to use remote procedural calls (RPC) to allow the programs to send messages back and forth. I intend on handling these tasks using the subprocess and xmlrpc libraries, respectively.

I will have a base “RpcThread” class, which I will derive my other thread classes from, and each thread class will have its own service for handling the requests it gets.



Since my simulator is communicating with multiple different programs, I need to run several threads asynchronously to handle multiple different xmlrpc connections simultaneously. I intend on using the threading library to do this. However, other than being able to handle multiple xmlrpc connections at once, asynchronous code execution is undesirable, so I will use event gates to regulate the flow of the program and ensure that during the simulation, only one thread is running at any given moment.

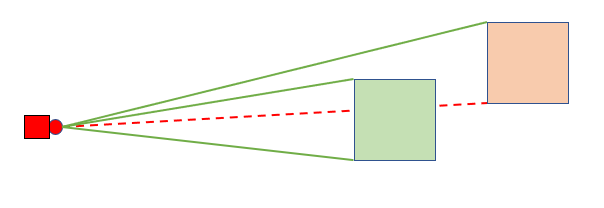
This diagram illustrates how the flow of execution will need to move between threads and processes so as to ensure that the simulation maintains a coherent notion of “the current time”.



### Modelling Vision

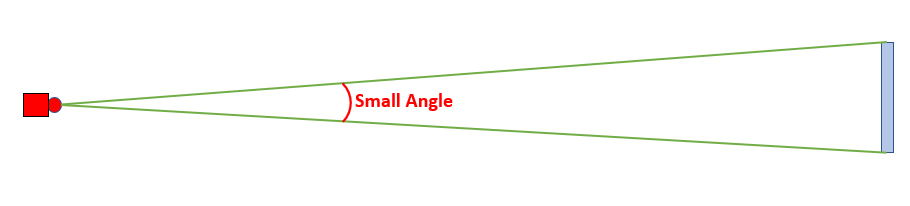
Determining the visibility of markers will be done using vector calculations. All obstructions are represented as a set of bounded planes, and of the markers within the field of view of the camera, the marker is visible if the following two conditions are met:

The lines between the camera and all four corners are not obstructed by any of the obstruction planes. This can be calculated by finding the point where the line intersects the unbounded plane, and then seeing if that point lies both in front of the corner, and within the bounds of the obstruction plane.



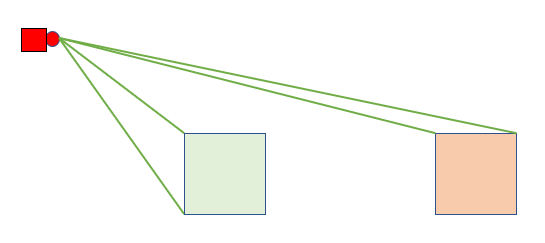
In this birds-eye view, the front face of the green square is visible, as the lines between its corners and the camera are unobstructed. However, the front face of the orange square is not visible, as the line between one of the corners of the square is obstructed.

The angle between two lines is not too small so as to not cover a large enough portion of the “image”:



In this birds eye view, the angle between two of the lines is too small, so the blue marker would not be visible.

It is also worth noting that due to the 3D nature of the simulation, it is possible to see “over” a token, and see the marker on top of a token.



In this side view, both squares are visible, as the camera sees “over” the token.

# **Documented Design**

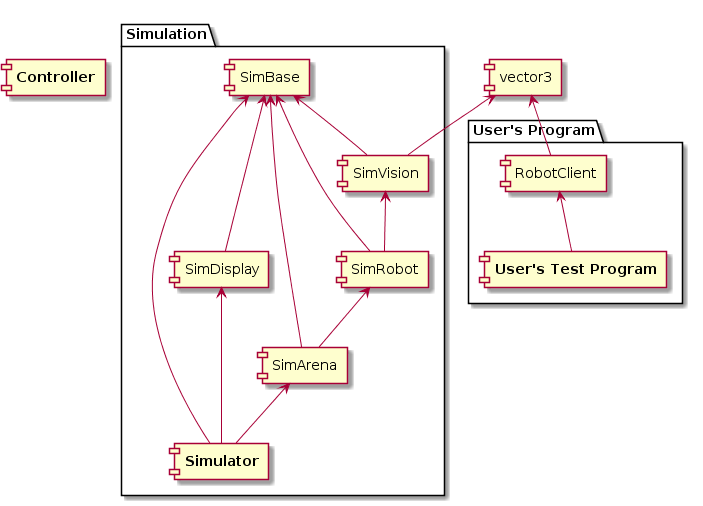
My program is written in python 3, and uses the subprocess library to run programs written by the user. These programs use a module that I wrote which emulates the API of the real robot module, and this replacement module speaks to my main “Simulator” program using the xmlrpc library. As explained in section 2.4.1, the simulator program needs one thread for each connection, plus one main thread for the simulation (this is done using the threading library). The simulator coordinates the processes and threads, ensuring that only one process is running at any given time, and provides a simulated arena for the robots to drive in.

The simulation itself uses the pymunk library to calculate 2D physics in the arena. My vector3 and SimVision modules perform vector calculations to identify what objects are visible to a robot’s camera. Finally, the pygame library is used to render a bird’s eye view of the arena for the user to watch their program run in.

## Source Files and Module Organisation

My program is split across 8 files (excluding configuration and robot programs), of which Simulator and Controller are both processes, and the remaining 6 files are modules imported by them. Also of note is the user’s program, which isn’t part of my project but makes use of my modules.

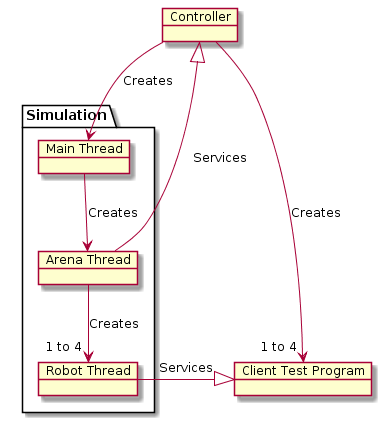
Arrows point towards dependencies, so SimVision imports vector3.



A brief overview of the contents of each file is provided below.

* **Controller.py** is the program that creates and handles all the other processes. It is also responsible for the Arena Thread, though mostly only uses it as a means of communicating with the Simulator.
* **Simulator.py** is the program that handles the simulation, and all the RPC servers for inter-process communication.
* **SimBase.py** is a module imported by everything else in the simulation process. It contains global variables, the pymunk derived classes for the objects in the simulation, and a base class for RPC threads.
* **SimArena.py** contains the thread class and associated RPC service class for the arena object, which the controller uses to interface with the simulation.
* **SimRobot.py** similarly is the thread and service class for the robots, which the client processes use.
* **SimVision.py** is a module that handles all of the complexities of the vision system, for use by the robot service.
* **SimDisplay.py** is a module that uses pygame to draw the display, and let the user close the simulation prematurely.
* **RobotClient.py** is a substitute module for the actual “robot” module used on robot hardware. It presents the same API.
* **vector3.py** is a module containing classes and functions for 3D vectors and bounded planes. It is used by SimVision and RobotClient.

When running, my program creates up to 5 additional processes, and runs up to 11 threads. I detail how they’re coordinated further down, but the threads and processes are created as follows:



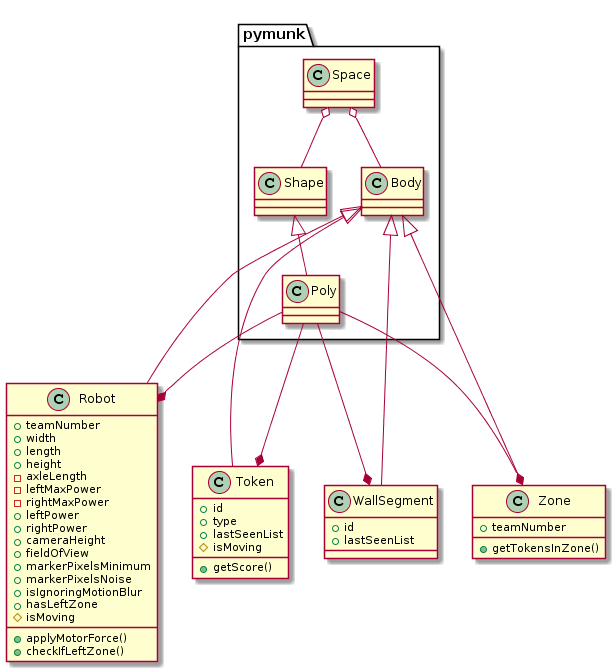
A brief overview of the purpose of each thread is provided below:

* The **Controller** process creates all the other processes, captures their output into a single channel, and cleans them up at the end.
* The **Simulator Main Thread** handles the actual simulation of the competition, and coordinates the other threads and processes to ensure they run only at the appropriate times.
* The **Simulator Arena Thread** presents an RPC interface to let the Controller set up the arena, add robots and log output during and at the end of the run.
* The **Simulator Robot Thread** presents an RPC interface to let a Client Test Program interact with its simulated robot.
* The **Client Test Program** is the program written by the end user that they want to test.

## Object Diagrams

All of the classes for my “physical” objects in the simulation are **derived** from the pymunk.Body class, and **contain** a single Poly shape which describes their size and physical attributes. These classes are declared in the SimBase file at from lines 108 onwards.

In the diagram below, a green circle indicates a public attribute or method, an orange diamond indicates a read only property, and a red square indicates a private attribute.



## Data

### Global Variables

These are global to all modules of the Simulator and kept inside the SimBase module (SimBase.py lines 9 to 26).

|  |  |  |  |
| --- | --- | --- | --- |
| Variable | Purpose | Written By | Read By |
| space | The pymunk space that the simulation takes place in. Initially set to “None”, and initialised properly when the ArenaService is started.  This is used for almost every simulation related operation, but its most common applications are finding bodies in the arena or a section of it, and adding new bodies to the simulation. | Simulator, SimBase, SimArena | SimBase, SimVision, SimDisplay, SimRobot |
| theTime | The amount of simulated time elapsed since the beginning of the simulation.  Simulator increases it every step, and other programs typically read from it to append timestamps to things. | Simulator | Simulator, SimBase,  SimRobot, SimVision, SimDisplay |
| endTime | The duration of the competition.  Set to 180 during startup, and only changed by SimDisplay if it receives a user input to end the simulation early. Only directly used in the SimBase.isSimulationRunning() function, which checks if theTime has passed endTime. | SimDisplay | SimBase |
| mainGate | A threading event used to block the main thread.  Only Simulator accesses this directly, though it is worth noting that all threads modify it through use of functions in the base rpcThread class from SimBase. | Simulator, SimBase | Simulator |
| rpcThreads | A list of all the threads in the simulator created for the purpose of handling xmlrpc connections.  Populated by Simulator and SimArena as threads are created, and iterated through when checking to see if all the threads are ready to start, or one is ready to make its next move. | Simulator,  SimArena | Simulator,  SimBase,  SimArena, |
| pendingOutput | A list of all messages for the controller to print the next time it requests the output of the simulator.  SimRobot appends items to the list when it receives a print request, and SimArena reads and clears it when it receives a request from the Controller. | SimArena | SimArena, SimRobot |
| wallSegments | A list of all WallSegment bodies in the arena.  Populated by SimBase when segments are created – this isn’t currently used by anything, though I’ve kept it for symmetry. | SimBase |  |
| tokens | A list of all Token bodies in the arena.  Populated by SimBase when tokens are created, and read from when the Arena iterates through every token to calculate scores. | SimBase | SimArena |
| robots | A list of all Robot bodies in the arena.  Populated by SimBase when robots are created, and read from when the Arena calculates scores and iterates through every robot to check if it is valid for the “moved out of zone” point. | SimBase | SimArena |
| zones | A list of all Zone bodies in the arena.  Populated by SimBase when zones are created, and read from when a robot checks to see if it has left its zone. | SimBase | SimBase |

### Configuration Files

The user has access to a number of configuration files, which contain data stored using JSON that the user can change. These files are listed in my appendix.

JSON is used primarily as a bridge between human and computer readable programs. JSON is easy for a human to understand and edit, and using the json library, it is easy to read and process the contents of the file in the program.

The robot configuration is read starting from line 118 in SimBase.py, and the token configuration is read in SimArena.py, starting at line 53.

All input values are sanitised using SimBase.sanitiseInput (SimBase.py lines 39-54) for validity so the user cannot crash the simulator by entering an out of range value or one of the wrong type. If an input value is outside the acceptable range, it is changed to the closest acceptable value, and if it is the wrong type it is set to a default value.

There are two areas which allow randomness or noise in the simulated behaviour:

The “Motor Noise Range” parameter allows for a small asymmetry in motor performance. Each motor’s maximum power level is offset by a random amount when the robot is created, and this is then fixed for the duration of the simulation.

The “Marker Pixels Noise Range” parameter allows for randomness in the resolvability of markers. It is worth noting that if both the noise range and base threshold are set to 0, the robot has perfect vision and can see any marker in its field of view.

## Algorithms

### Multi-Threading and Processing

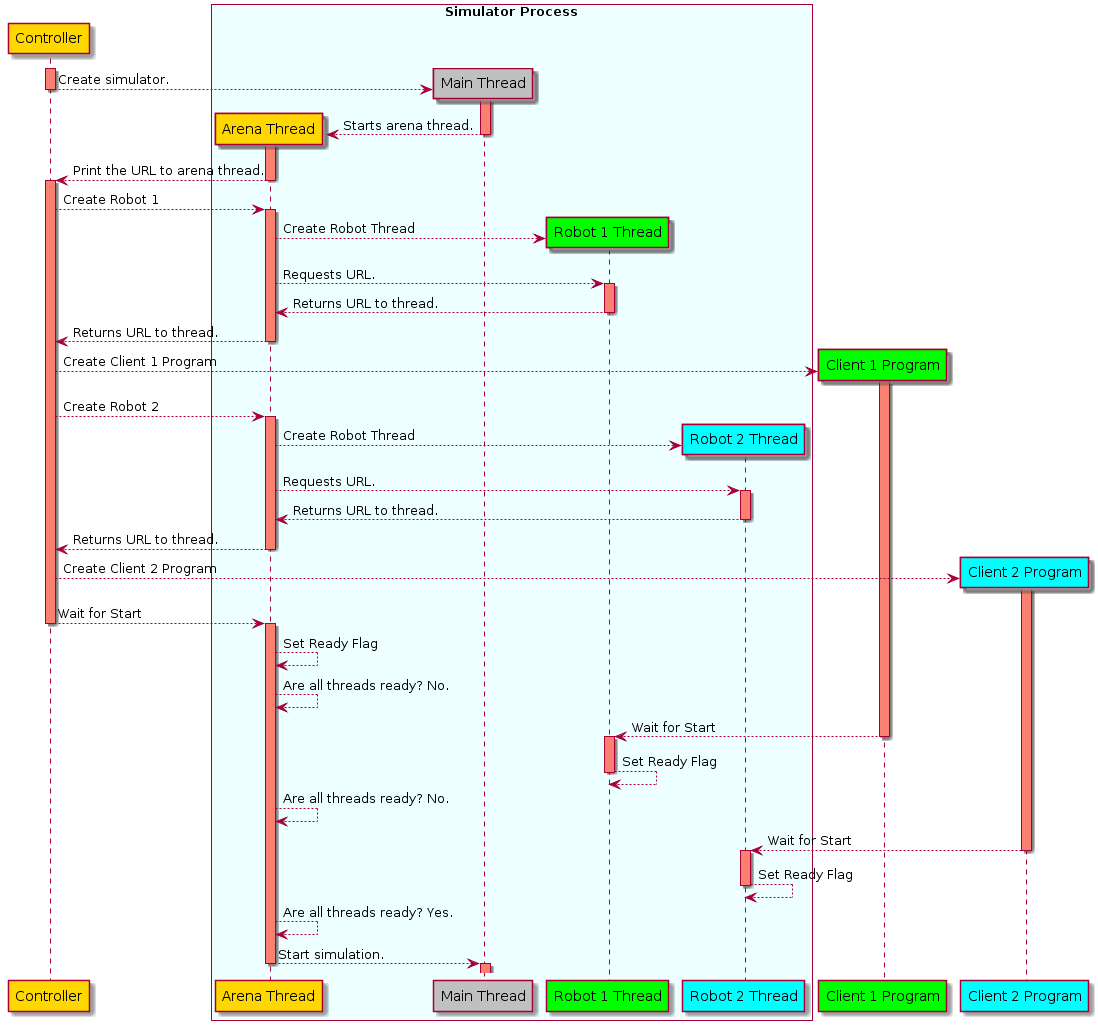
#### Motivation

In order to run arbitrary python programs written for robots, I need to run the program as a separate process and intercept calls to the robot (as writing my own python interpreter would be vastly beyond the scope of A Level). I can send information **to** the sub-process through its command line arguments or through Standard Input, and there are some limited ways to send information the other way such as by printing to Standard Output. However, these channels are insufficient to support complex two-way communication.

Instead, I use the xmlrpc library to send remote procedure calls between programs. This allows me to store a connection to a server, then when I call functions on that “connection” object, the corresponding function in the server program will be executed and return a value (or an exception) to the client. This allows me to give all four robot programs their own process, and handle all the shared interactions in a fifth “simulation” process.

The server side of an xmlrpc connection is provided by a function called serve\_forever() (for an example, see SimArena line 181), which does not return until the connection is shut down at the end of the session. This means that the thread which calls serve\_forever() cannot do anything else, so each connection needs a dedicated thread. The main simulator program also needs its own thread. This is done using the threading library.

#### Startup Sequence

The program starts with just the Controller running, with a list of robot programs to simulate, and the objective is to finish with all processes started, and all threads blocked except the Simulator’s Main Thread, which then enters its main loop. 

First, the Controller starts the Simulator process. When the Simulator is started, the Main Thread creates the Arena Thread and then immediately blocks itself. The Arena Thread starts its service, and prints the URL to the Standard Output. Unlike starting the other threads, there is no pre-existing RPC connection between the controller and simulator, so instead the Standard Output of the Simulator is captured by the Controller and parsed to extract the URL. This allows the Controller to establish the RPC connection with the Arena Service in the Simulator Process.

Next, the Controller tells the Arena Service to create a Robot Thread and its associated robot object. The Arena Service returns the URL of the new Robot Service to the Controller. The Controller then starts a new Client Program subprocess, providing the URL as a command line argument. The RobotClient library (which is imported by the Client Program) extracts the URL and connects to the Robot Service in the Simulator and calls a function to wait for the start of the competition. At this point, both the client and the robot thread are now blocked, and flagged as ready to start.

The Controller can repeat this process for as many other client programs are to be Simulated. It does not wait for any of them to be ready before starting the next one, which means up to four robots may be starting in parallel. However, once all robots have been started, it asks the Arena Service to wait for all the robots to be ready.

At this point, the Controller is now blocked and is waiting for the simulation to begin. Also, the client robots are starting and will soon be blocked, waiting for the simulation to begin. The Arena Thread is running in a loop, polling the ready status of the robots once every second. When they are all ready, the Arena Thread returns control to the Main Thread and blocks itself. This means all threads are blocked, and the Main Thread will now enter the main loop of the simulation, so startup is complete.

#### Main Loop Sequence

The main loop has four essential tasks to perform every iteration:

1. If any thread is scheduled to wake up at this time, it is unblocked and allowed to run until it next wants to sleep.
2. All robots have their motor forces applied, and a flag is set if they have left their zone.
3. pymunk steps the space forwards 1/64th of a second.
4. The display is redrawn, and any user input (such as closing the simulation) is processed.

Only the first of these steps involves thread scheduling. When the Arena Thread or a Robot Thread is ready to wake up, the Main Thread calls rpcThread.unblock(), which performs the following actions:

* Closes the gate of the main thread (though doesn’t stop it running).
* Opens the gate of the target thread (letting it run).
* Stop the main thread until the gate is opened.

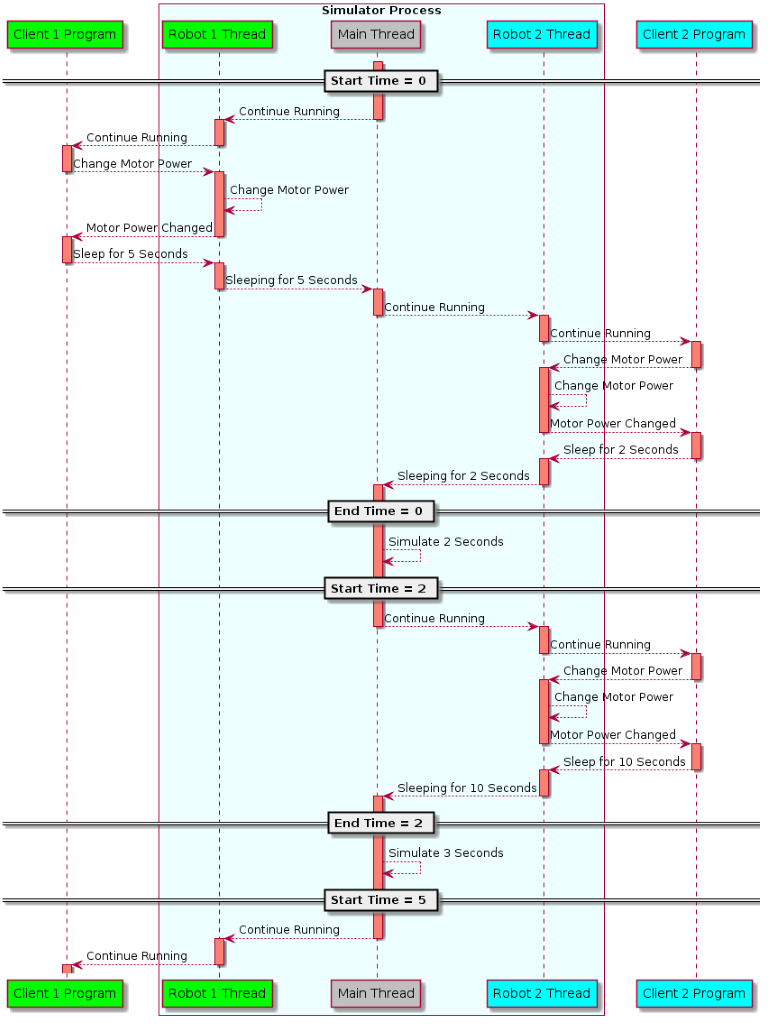
This effectively transfers the flow of control from the main thread to the target thread. When the target thread wants to return control to the main thread, it calls rpcThread.block(), which performs a similar set of actions but closes its own gate and opens the gate of the main thread.

When the target thread is unblocked, it continues from where it was before. In almost all cases, this was processing an incoming RPC call from its Client Program or the Controller. Consequently, in almost all cases, the next thing it does is return to that process, and the entire Simulator is blocked. If the client process crashes, gets stuck in a loop, or stops at this point, the Simulator will never be unblocked and will hang indefinitely. This is a known defect of the design and is discussed in the evaluation section 4.2.1.

If the Client Process does issue another RPC call – that call may be handled entirely within the RPC thread, and not involve the main loop, or it may sleep and return to the main loop. When it sleeps, it updates its wakeup time to a time later in the simulation.

In this example diagram, at the very start of the simulation, all the threads are ready to wake up, and it does not matter which order they are allowed to run because the simulation will not advance in time until they have all performed their initial actions.

Its worth noting that during the simulation, the MainThread also returns to the Arena Thread 8 times a second to give the Controller a list of all the print statements that have accumulated since the last time, which it prints to its Standard Output. This has been omitted from the diagram for space reasons.



#### Shutdown Sequence

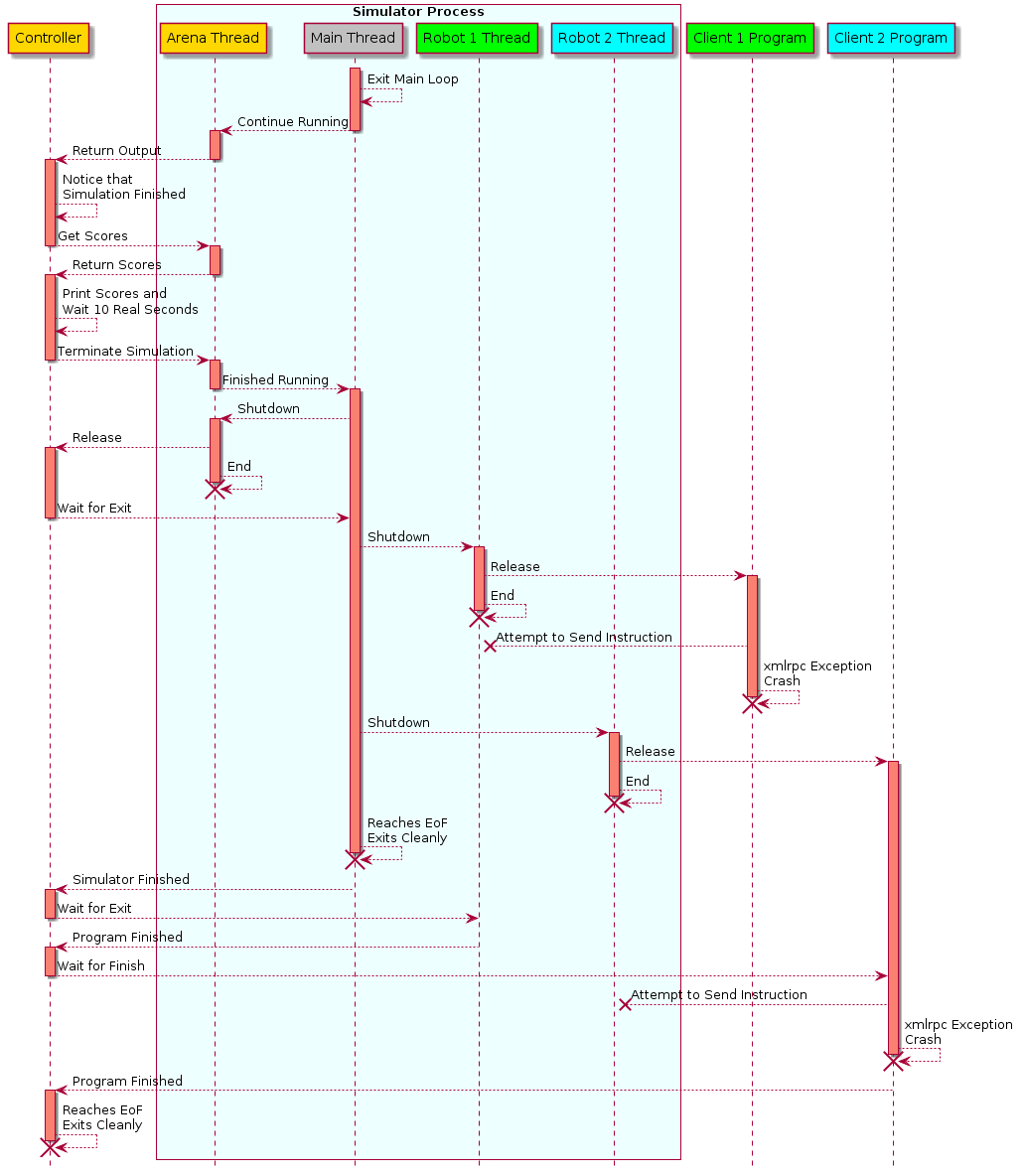
When the simulation reaches its duration, the Controller must print the scores of the competition, and all the processes (including the Simulator and Controller) must end.

The process begins with the Main Thread exiting the main loop and unblocking the Arena Thread, causing it to immediately return the Controller’s current call to waitForOutput with a flag to indicate that the simulation is over.

At this stage, everything is still running, so the Controller can ask the Arena Service to calculate the final scores. It then waits 10 seconds to let the user examine the final state of the competition, and then asks the Arena Service to initiate the final shutdown. This is done by blocking the Arena Thread and releasing the Main Thread, which will tell all the RPC threads to shut down, and exit cleanly as it reaches the end of the program.

In doing this, the Simulator unblocks the processes attached to the threads – the Controller will next begin waiting for all processes to finish, and the Client Programs will continue running until they attempt to call a robot function, at which point they will crash because the RPC server they were connected to is gone. This crash is ok, because I want these processes to terminate anyway, although wrapping them in a Context Manager would allow me to do it more cleanly.

Once the controller has finished waiting for all other subprocesses to finish, it will also finish, leaving no processes running.



### Vision

In a real robot, when the program calls *see()*, the camera takes a picture and the program yields briefly as the image is processed. It then returns a list of “Marker” objects for every marker it could identify in the picture, which contain information about the marker, and information describing where each corner and the centre of the marker was seen, in several different coordinate systems for the convenience of the programmer.

From the point of view of my simulator, half of this job will be done on the server side, and the other half will be done by the client side module. Working out which objects are visible will be done on the server, as that has access to the positions and orientations of every object in the simulation. However, putting them into a “Marker” structure will need to be done by the client program’s module, as I cannot send arbitrary objects through xmlrpc, and the client program needs to be able to see the class definitions anyway.

Working out which items are obstructed is done using a large amount of vector arithmetic, all of which is covered in my Further Maths A Level. Objects are represented as a collection of bounded planes, and if a line intersects by one of these planes, it is blocked. If the line between the camera and a point is not blocked by any plane, the point is visible.

#### vector3.py

Since I am doing a lot of work with vectors, I created a class for 3D vectors. It was also useful to define a class for bounded planes, since those calculations are complex and specific to this task. The class definitions for both of these are found in vector3.py.

##### Class Vector3

My Vector3 class has 3 attributes and 2 properties – all three components of the vector are attributes (and defined on creation), and the magnitude and unit vector of that vector are both accessible as read only properties.

The class has overloaded the pythonic functions for =, +, -, \*, and str(), allowing 2 vectors to be compared or added/subtracted to each other, and a single vector to be multiplied by a scalar or represented as a string.  
The class also has functions for finding the dot or cross product of two vectors, the angle in radians between two vectors, and a function to rotate a vector3 around the Z axis.  
Finally, a Vector3 can be converted into a dictionary of three floats (with the keys x, y and z), and reconstructed from a dictionary in this format. This is used to send Vector3s through an xmlrpc connection, which cannot send arbitrary objects.

##### Class Plane

My Plane class is constructed from three vectors, representing the bottom left, bottom right and top left corners of each point of the bounded plane in world space.  
It has no attributes, but the three vectors, the normal to the plane and the four parameters of the plane in cartesian form are all accessible as read only properties.

The primary purpose of the Plane class is its two helper functions:

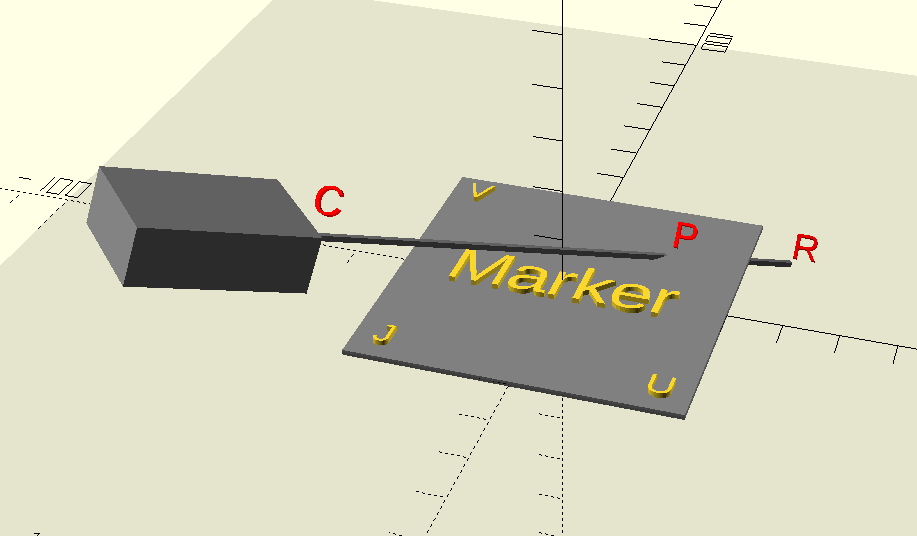
* isFacingCamera calculates if the plane is facing the camera, and while it is only one line long, it is useful as a more descriptive name for a mathematical expression.
* isObstructingPoint calculates if the bounded plane obstructs the line between the camera and a second point. It is detailed further in the section below.

##### Plane.isObstructingPoint()

This function is implemented in vector3, in lines 144 to 164.

To work out if the line between two points is obstructed, I perform a number of vector calculations. The diagram below shows the vectors used:

* **C** is the position of the camera, in world space.
* **R** is the position of the point being checked, in world space.
* **P** is the point of intersection between the line **CR** and the unbounded plane, in world space.
* **J** is the bottom left point on the bounded plane, in world space.
* **U** is the offset of the bottom right point on the bounded plane from **J**.
  + This could also be thought of as the position of the bottom right point in the local space of **J**.
* **V** is the offset of the top left point on the bounded plane from **J**.
  + As with **U**, this can also be thought of in terms of the local space of **J**.
* **N** (not shown on diagram) is the normal to the plane to the diagram, and is obtained as **U**×**V**.



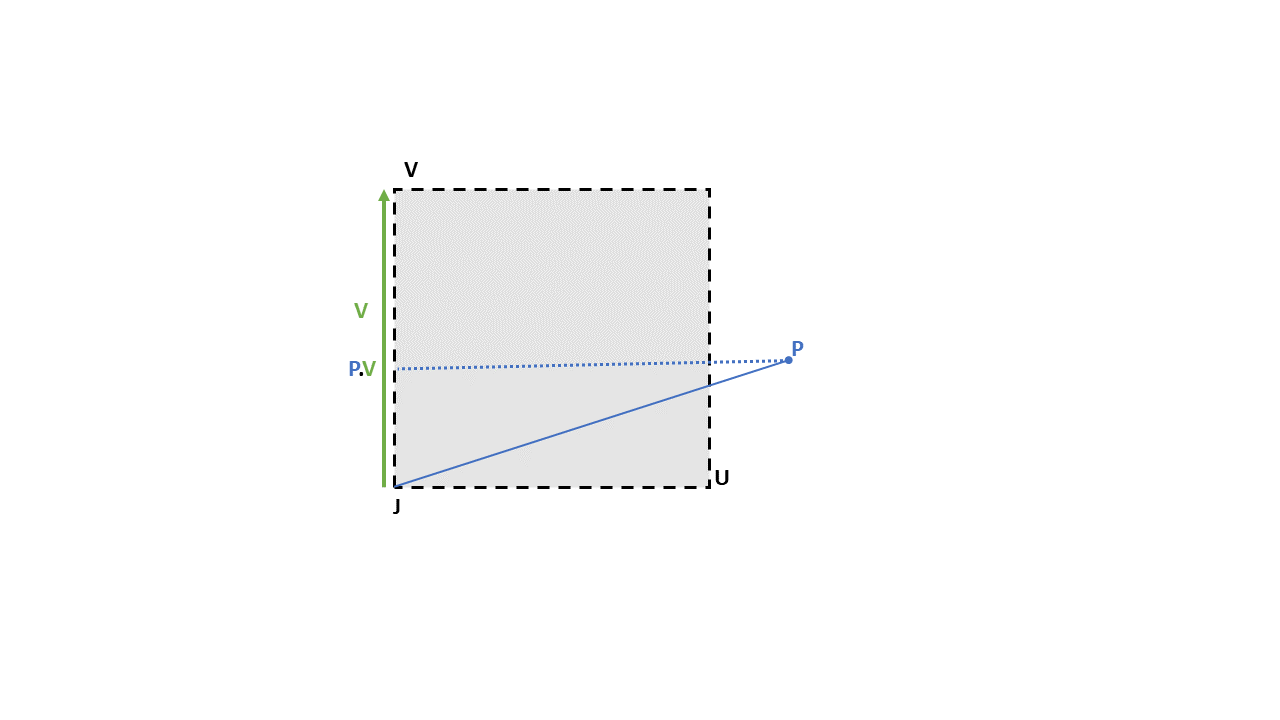
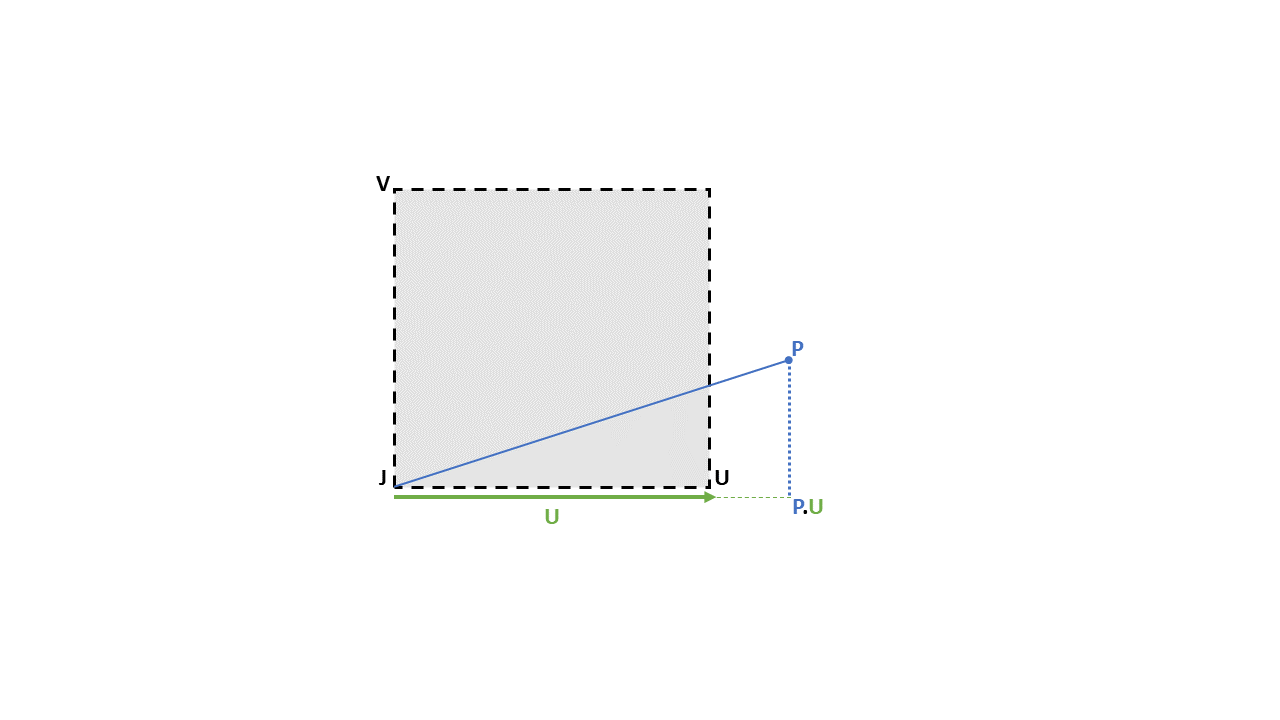
First, **P** is calculated. For any plane and line, unless the line and plane are parallel (in which case the plane cannot block the line and the function returns false), there is a point of intersection. The magnitude of **CP** over the magnitude of **CR** is found by the formula:

Where a, b, c and d are the coefficients of the cartesian equation of the Plane. In terms of the vectors they are:



If λ > 1, the unbounded plane intersects the line behind the point **R**, so the plane therefore cannot obstruct the line **CR**.  
If λ < 0, the unbounded plane intersects the line behind the camera **C**, so the plane therefore cannot obstruct the line **CR**.

If 0 < λ < 1, the unbounded plane intersects the line **CR** between the camera **C** and point **R**, so two similar calculations must be performed to find if the point of intersection lies within the bounds of the plane.



As with λ, if both and are between 0 and 1, then the bounded plane obstructs the line between the camera and the point.

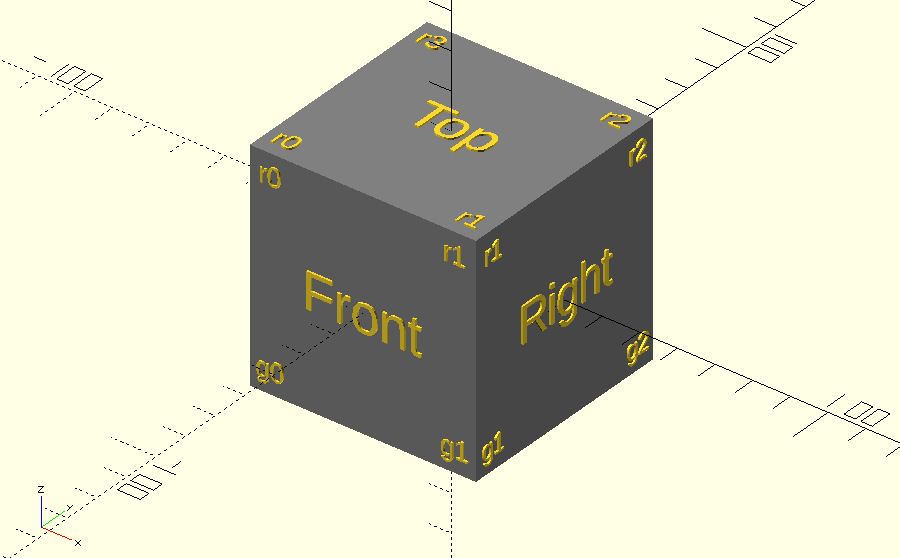
#### SimVision.py

With the Vector3 and Plane class handling most of the arithmetic, the SimVision module is mostly concerned with tracking what can be seen and what can obstruct objects.

##### \_getVisibleCuboidFaces()

One of the more notable functions is \_getVisibleCuboidFaces(), which returns a list of every face (represented as a Plane) that can be seen by the camera. It is defined on lines 7 to 34.

The 8 vertices of a cuboid are stored as ground vertexes 0 to 4, and raised vertexes 0 to 4, and (from the top down), the numbers increase going counterclockwise. For each face, the corners are ordered so as to ensure the normal to the face points out of the cuboid.



An attempt to reduce the number of unnecessary calculations is made here, as in a cuboid, some faces are totally obstructed by other faces, so it is impossible for such a face to be the only face obstructing a point. The bottom face is never visible, so the function ignores it entirely, and the function will check each other face to find if it is facing the camera, and only return it if so.

##### The Two Loops of the See() Function

The main vision code has two loops – first, it constructs a list of every body with visible markers, and a list of every plane that could potentially obstruct a line. The function as a whole begins on line 127, and continues to line 188. The first loop starts on line 143, and ends on line 152. The second loop is on lines 153 to 176.

Below is the loop that constructs a list of every obstructing plane and every body with markers on. Notably, the markers on tokens cannot be seen if the token is moving (though the token can still obstruct things), and wall segments cannot be potential obstructions (because nothing can be behind one).

**for** each body in the space:  
 **if** it is another robot:  
 Add the visible faces of the body to the list of potential obstructions.  
 **if** it is a token:  
 Add the visible faces of the body to the list of potential obstructions.  
 **if** it is not moving:  
 Add it to the list of bodies with markers on.  
 **if** it is a wall segment:  
 Add it to the list of bodies with markers on.

To work out which markers are visible, the program checks to see if the plane is resolvable, then checks every corner to see if it is within the field of view and unobstructed. If none of the corners are obstructed, the marker is visible.

**for** each body with markers on:  
 Get all the sets of marker corners.  
 **for** each set of marker corners:  
 **if** the plane is not resolvable: skip it.  
 **for** each corner:  
 **if** it is outside the field of view: skip it. #see text below  
 **for** each potential obstruction:  
 **if** it obstructs the corner: skip the corner.  
 **if** no corners were skipped:  
 Add the marker to the list of visible markers.

To simplify the calculations, when calculating if a marker is outside the field of view, the field of view is approximated to a cone (rather than a rectangular pyramid). This is not perfectly accurate, as the extreme corners of the rectangular field of view are outside the cone, and the top and bottom segments of the cone are outside the rectangular field of view.  
However, for a reasonably mounted camera, it is very unlikely for a marker corner to reside in these areas, and vision with the real hardware is not completely reliable, so the approximation does not have a significant impact.

Finally, it is worth noting that bodies have a list of when they were last seen, with four entries (one for each robot). That list is updated in this loop on line 175, and used by the display module (section 2.5.2) to identify markers.  
Not only is this one of the requirements (3c), but it is also very useful for testing.

#### RobotClient.py

The server cannot return the markers in the required object structure, as xmlrpc cannot send objects of user-defined classes. As a result, the server returns a dictionary, containing all the information needed by the RobotClient to construct a list of Marker objects (see SimVision.py lines 179 - 188).

The dictionary contains a number of values which are the same for all markers:

* Resolution the image was taken at.
* Field of view of the camera.
* The position of the camera in world space.
* The direction of the camera.
* The time at which the image was taken.

It also contains a list of more dictionaries, each entry containing the information needed to construct one marker object:

* A list of the four corner positions (in world space).
* The id of the marker.
* The size of the marker.

In RobotClient, the corresponding unpacking is done on lines 179 to 189 – the helper functions and classes called by this span lines 11 to 117. Most of the data just needs to be rearranged into the expected format (as specified by <https://hr-robocon.org/docs/vision.html>). The coordinates come in as dictionaries (which are converted back into Vector3s before any calculations are done), and they are relative to the Simulator’s world coordinate system. The Marker object must return them in cartesian coordinates relative to the camera, and with different axis definitions (X is rightwards, Y is downwards, and Z is forwards). This conversion can be done using vector arithmetic (since the direction of the camera is known).

Having transferred the positions into camera space, the Marker object also returns them as polar coordinates, which is calculated by turning the X and Y components of the positions into angles.  
Finally, the Marker object also returns the image pixel coordinates of the points, so the polar coordinates are scaled by the field of view and resolution to get the image pixel coordinates.

## User Interface

The user interface of the simulation has two parts – the simulation is started from the command line, but then displayed graphically in a separate window.

### Command Line Syntax

To start the simulation, the Controller.py program must be started with a list of each robot to test (in order) as arguments. (If the program is started without any arguments, it will still run a simulation, but without any robots.)

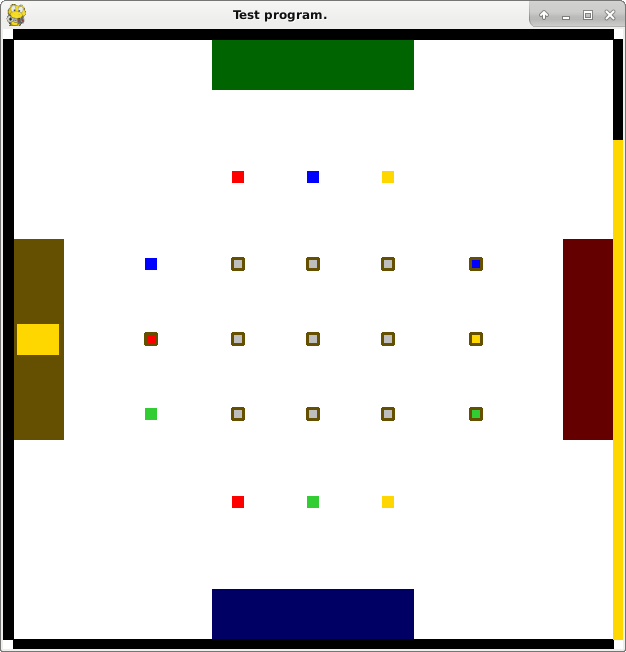
The command line syntax is therefore:

$ python3 Controller.py --test testProgram1.py --test testProgram2.py

Up to four test programs are allowed. Any additional ones are ignored.

### SimDisplay.py

The display shows the walls, zones and robots, and the different categories of token. Wall Segments are highlighted if they were seen in the last second, using the colour of the robot that last saw them. Similarly, tokens are highlighted with a dark border if they have been seen within the last second.



The simulation can be ended prematurely by closing the display window or pressing the Escape key. When the simulation ends, the window displays the final state of the simulation for a further 10 seconds for further inspection before the window is removed.

# **Testing**

Different test strategies are appropriate for different parts of the code. Unit testing and component testing are appropriate for details of the implementation, and broader system testing is more appropriate for the end user requirements. The following list indicates where each requirement was tested.

1. Tested in **End User Level Testing**.
2. Tested in **System Testing** and 2b is tested in **Testing Other Requirements**.
3. All tests implicitly test this. 3c is tested in **SimVision Component Test**.
4. Tested in **End User Level Testing**.
5. Tested in **System Testing**.
6. Tested in **Other Requirements** and **System Testing**.
7. Tested in **Other Requirements**.
8. Tested in **Vision Testing**.
9. Implicitly tested in order to set up test cases in **Vision Testing** and **Other Requirements**.
10. Was not implemented.

## Flow Control Testing

These tests show that the flow of the program is as defined by the sequence diagrams in the “Multi-Threading and Processing” section of the design document.

Trace statements have been added before and after every point where a thread or child process is started or stopped, and every point where a gate is set or waited on. A few other traces have been added in other places to further clarify the flow of the program.

### Verifying the Startup Sequence:

As shown by this trace output, the initialisation process follows the expected sequence of events.

It is worth noting that when the robot processes are started, they briefly run in parallel alongside the arena thread. This is occurring from lines 16 to 27, though by chance only it disrupts lines 19 to 26 – this is not a bug, as the arena thread does not allow the simulation to begin until all the robots have finished their initialisation and blocked themselves, so are ready to run sequentially.

1. **In Controller: Starting simulator process.**
2. **In Controller: Started simulator, waiting for URL.**
3. **In MainThread at 0: Simulator starting.**
4. **In MainThread at 0: Creating ArenaThread.**
5. **In MainThread at 0: Starting ArenaThread.**
6. **In MainThread at 0: Simulator is waiting for clients to be ready to begin.**
7. **In Controller: URL received, connecting to the arena service.**
8. **In Controller: Connected to arena service.**
9. **In Controller: Creating robot number 0**
10. **In Controller: Robot created, starting test program subprocess.**
11. **In Controller: Test program subprocess created.**
12. **In Controller: Creating robot number 1**
13. **In Controller: Robot created, starting test program subprocess.**
14. **In Controller: Test program subprocess created.**
15. **In Controller: All robots created, waiting for start.**
16. In [0]-Thread at 0: Entering ArenaService.waitForStart()
17. In [0]-Thread at 0: Arena is waiting for [0]-Thread
18. In [0]-Thread at 0: Arena is waiting for [1]-Thread
19. **LazyRobot started!**
20. **In RobotClient: Connecting to RobotService:**
21. **LazyRobot started!**
22. **In RobotClient: Connecting to RobotService:**
23. **In RobotClient: Connected, waiting for start.**
24. In [1]-Thread at 0: Robot waiting for start.
25. **In RobotClient: Connected, waiting for start.**
26. In [2]-Thread at 0: Robot waiting for start.
27. In [0]-Thread at 0: Arena is waiting for [2]-Thread
28. In [0]-Thread at 0: All robots ready, leaving ArenaService.waitForStart()
29. In [0]-Thread at 0: Yielding control to MainThread
30. **In MainThread at 0: All clients are ready to begin, entering main loop.**

### Verifying the Main Loop Sequence:

This trace shows that during the main loop, only one thread (and its associated process) is running at any given time. All colour blocks begin with either “Robot now starting” (for unblocked for the first time) or “Receiving control from the main thread” (for being unblocked subsequent times).

1. **In MainThread at 0: All clients are ready to begin, entering main loop.**
2. In [0]-Thread at 0: Receiving control from MainThread
3. **In Controller: Receiving control from Simulator.**
4. **In Controller: Yielding control to Simulator.**
5. In [0]-Thread at 0: Entering ArenaService.waitForOutput()
6. In [0]-Thread at 0: Yielding control to MainThread
7. In [1]-Thread at 0: Robot now starting.
8. **In RobotClient: Starting.**
9. **In RobotClient: Entering sleep.**
10. In [1]-Thread at 0: Entering RobotService.sleep()
11. In [1]-Thread at 0: Yielding control to MainThread
12. In [2]-Thread at 0: Robot now starting.
13. **In RobotClient: Starting.**
14. **In RobotClient: Entering sleep.**
15. In [2]-Thread at 0: Entering RobotService.sleep()
16. In [2]-Thread at 0: Yielding control to MainThread
17. In [2]-Thread at 6.0: Receiving control from MainThread
18. In [2]-Thread at 6.0: Exiting RobotService.sleep()
19. **In RobotClient: Exiting sleep.**
20. **In RobotClient: Entering sleep.**
21. In [2]-Thread at 6.0: Entering RobotService.sleep()
22. In [2]-Thread at 6.0: Yielding control to MainThread
23. In [2]-Thread at 12.0: Receiving control from MainThread
24. In [2]-Thread at 12.0: Exiting RobotService.sleep()
25. **In RobotClient: Exiting sleep.**
26. **In RobotClient: Entering sleep.**
27. In [2]-Thread at 12.0: Entering RobotService.sleep()
28. In [2]-Thread at 12.0: Yielding control to MainThread
29. In [1]-Thread at 18.0: Receiving control from MainThread
30. In [1]-Thread at 18.0: Exiting RobotService.sleep()
31. **In RobotClient: Exiting sleep.**
32. **In RobotClient: Entering sleep.**
33. In [1]-Thread at 18.0: Entering RobotService.sleep()
34. In [1]-Thread at 18.0: Yielding control to MainThread
35. In [2]-Thread at 21.0: Receiving control from MainThread
36. In [2]-Thread at 21.0: Exiting RobotService.sleep()
37. **In RobotClient: Exiting sleep.**
38. **In RobotClient: Entering sleep.**
39. In [2]-Thread at 21.0: Entering RobotService.sleep()
40. In [2]-Thread at 21.0: Yielding control to MainThread
41. In [0]-Thread at 30.0: Receiving control from MainThread
42. In [0]-Thread at 30.0: Exiting ArenaService.waitForOutput()
43. **In Controller: Receiving control from Simulator.**
44. **In Controller: Yielding control to Simulator.**
45. In [0]-Thread at 30.0: Entering ArenaService.waitForOutput()
46. In [0]-Thread at 30.0: Yielding control to MainThread

### Verifying the Shutdown Sequence:

This trace output also shows that the shutdown sequence is being followed, although due to the parallel running of almost every process during the shutdown sequence, their traces are heavily interleaved in places.

Multiple threads and processes begin being run in parallel starting from line 11, when the main thread starts to shut down all the other threads in the simulator. Both robot programs attempt to sleep and crash at almost exactly the same time (lines 28 and 29), and their stack tracebacks overlap one another in the actual trace.

The parallel execution ends on line 31, when the simulator process ends, and the controller is the only process left running.

1. **In MainThread at 180.0: Yielding control to [0]-Thread**
2. In [0]-Thread at 180.0: Receiving control from MainThread
3. In [0]-Thread at 180.0: Exiting ArenaService.waitForOutput()
4. **In Controller: Receiving control from Simulator.**
5. **In Controller: Yielding control to Simulator.**
6. In [0]-Thread at 180.0: Entering ArenaService.waitForOutput()
7. In [0]-Thread at 180.0: Exiting ArenaService.waitForOutput()
8. **In Controller: Receiving control from Simulator.**
9. **In Controller: Simulation no longer running. Calculating scores.**
10. **In Controller: Scores calculated, yielding control to Simulator.**
11. In [0]-Thread at 180.0: Yielding control to MainThread
12. **In MainThread at 180.0: Releasing [0]-Thread to shut down.**
13. In [0]-Thread at 180.0: Receiving control from MainThread
14. **In MainThread at 180.0: [0]-Thread has shut down.**
15. **In MainThread at 180.0: Releasing [1]-Thread to shut down.**
16. In [1]-Thread at 180.0: Receiving control from MainThread
17. In [1]-Thread at 180.0: Exiting RobotService.sleep()
18. **In Controller: Receiving control from Simulator.**
19. **In MainThread at 180.0: [1]-Thread has shut down.**
20. **In Controller: Waiting for Simulator finish.**
21. **In MainThread at 180.0: Releasing [2]-Thread to shut down.**
22. In [2]-Thread at 180.0: Receiving control from MainThread
23. In [2]-Thread at 180.0: Exiting RobotService.sleep()
24. **In RobotClient: Exiting sleep.**
25. **In RobotClient: Entering sleep.**
26. **In RobotClient: Exiting sleep.**
27. **In RobotClient: Entering sleep.**
28. **Robot Program 1 Crashes. The Traceback has been omitted for space reasons.**
29. **Robot Program 2 Crashes. The Traceback has been omitted for space reasons.**
30. **In MainThread at 180.0: [2]-Thread has shut down.**
31. **In MainThread at 180.0: Simulator process ends**
32. **In Controller: Simulator has finished.**
33. **In Controller: Waiting for robot test program to finish.**
34. **In Controller: Robot test program has finished.**
35. **In Controller: Waiting for robot test program to finish.**
36. **In Controller: Robot test program has finished.**
37. **In Controller: All subprocesses have finished. Simulation successful.**

## Vision Testing

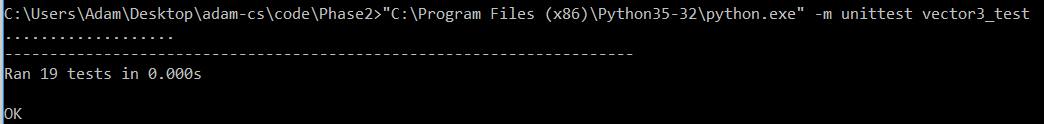
Vision has two components. They require different testing strategies.

### vector3 Unit Test

The first component is the vector3 module, which I unit tested to find typing errors and similar mistakes. This section was particularly amenable to unit testing as it does not have any dependencies itself, and unit testing was particularly important here because like all numerical code, it is easy for it to deliver the wrong number without any indication of an error.

I wrote the unit test file as I wrote the module, which meant that I identified issues as they came up. In addition to identifying a few typing errors, these tests also highlighted the need for a unary minus operator (-Vector), a reverse multiply operator (for doing k\*vector, as opposed to vector\*k), and a check for different types when checking for equivalence (so it would return False instead of crashing).

Having written the tests, I can run the unit test file using the command line below, and all 19 tests pass. These tests can all be found in vector3\_test.py in my Appendix.



### SimVision Component Test

The second component is the higher level SimVision module, for which I set up test cases and observed the resultant behaviour.

#### Description of Test Cases

These test cases are set up by changing the token position configuration (Token Position Config.json), and running a robot client program (TestProgramSee.py) which just sits in its starting position and calls the see() function. I defined the configuration parameters in Robot 0 Config.json.

Unless stated otherwise, robot configuration parameters are set as:

* The field of view is 45 degrees.
* The pixel threshold and noise are 0, so any unobstructed marker within the field of view is visible.
* The camera height is 0.05, so tokens will block all line of sight.

The walls were also removed from the simulation to simplify the output, unless otherwise stated.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Test | Purpose | Expected Outcome | Actual Outcome |
| 1 | A token is placed 1m in front of the robot. | To test that the robot can see markers. | Sees the front marker of the token. | **The marker was not visible.** |
|  | Repeat of 1. | To test my fix for the error the first time the program was run. | Sees the front marker of the token. | **As expected.** |
| 2 | A token is placed 1m behind the robot. | To test that markers behind the robot are not visible. | Does not see any token markers. | **As expected.** |
| 3 | A token is placed 1m in front of the token and 2m to the right. | To test that markers outside the field of view cannot be seen. | Does not see any token markers. | **As expected.** |
| 4 | A token is placed 1m in front of the robot, another token is placed directly behind it. | To test that markers can be obstructed by other bodies. | Only sees the front marker of the first token. | **As expected.** |
| 5 | As test 4, but the camera height is set to 0.2. | To test that it is possible to see “over” markers. | Sees the front marker of the first token, and the top markers of both tokens. | **As expected.** |
| 6 | As test 1, but the pixel threshold is set to 200. | To test that it is possible for a marker to be too far away to resolve. | Does not see any token markers. | **As expected.** |
| 7 | A token is placed so the front face is 1m in front of the camera, the camera height is 0.05, the field of view is 57 degrees, and all supporting data for the marker is checked. | To test that all the information returned by the marker object is correct. | As 1 – the expected supporting data is included alongside the actual supporting data, for space reasons. | **The orientation of the marker is incorrect.**  **The order of the corners is incorrect.**  **All other values are correct.** |
| 8 | The walls are placed in the arena, and the robot looks at a wall. | To test that the wall markers are visible. | See all wall markers in the field of view. | **As expected.** |
| 9 | As test 1, but another robot is placed between the camera and token. | To test that robots obstruct vision. | Does not see any token markers. | **As expected.** |
| 10 | All tokens and robot are placed in the arena as if it was an actual competition. | An overall sanity check for the vision system. | Sees all tokens and wall segments in the field of view that are not obstructed by another object. | **As expected.** |

#### Sample of Test Results

A border around a token indicates that at least one marker on it was seen, and a coloured wall segment indicates that its wall marker was seen.

|  |  |  |
| --- | --- | --- |
| **Test 1:** | **Test 2:**    Note that robot is facing right. | **Test 3:**  It is outside the field of view. |
| **Test 4:**  The camera is too low. | **Test 5:**    The camera is higher. | **Test 10:** |

#### Full Details of Test 7

For test 7, I positioned the camera at 5.5cm height, so that it points straight at the centre of the marker in front of it. The marker is 10cm at a distance of 1m, so it subtends a tenth of a radian. I set the camera field of view to 1 radian (57 degrees), so I expect the width of a marker to be 1/10th of the resolution of the camera (1920, 1440), and centred about the middle (960, 720).

The other values were calculated using trigonometry. Floating point decimals in the actual values have been cut down for space reasons.

|  |  |
| --- | --- |
| **Expected Values:**  Marker Id 32  Distance = 1  rot\_y = 0  Orientation = **(0, 0, 0)**  Bottom Left Vertex:  Image x Coordinate: 768 (= 960 - 192)  Image y Coordinate: 912 (= 720 + 192)  Polar Coordinates: 2.86, -2.86 World Coordinate : -0.05, 0.05, 1  Bottom Right Vertex:  Image x Coordinate: 1151  Image y Coordinate: 912  Polar Coordinates: -2.86, 2.86 World Coordinate : 0.05, -0.05, 1  Top Right Vertex: Image x Coordinate: 1151  Image y Coordinate: 528  Polar Coordinates: -2.86, 2.86 World Coordinate : 0.05, -0.05, 1  Top Left Vertex:  Image x Coordinate: 768  Image y Coordinate: 528  Polar Coordinates: -2.86, -2.86 World Coordinate : -0.05, -0.05, 1 | **Actual Values:**  Marker 32:  Vertex[0]: image=(1151.839501,911.8395013)  Vertex[0]: polar=(2.862405226,2.862405226)  Vertex[0]: world=(0.05,0.05,0.99999999999)  Vertex[1]: image=(768.1604986,911.8395013)  Vertex[1]: polar=(2.862405226,-2.86240521)  Vertex[1]: world=(-0.05,0.05,0.99999999999)  Vertex[2]: image=(768.16049,528.16049)  Vertex[2]: polar=(-2.86240522,-2.86240522)  Vertex[2]: world=(-0.05,-0.049999,0.99999)  Vertex[3]: image=(1151.8395,528.160498)  Vertex[3]: polar=(-2.86240522,2.86240522)  Vertex[3]: world=(0.05,-0.04999999,0.999999)  dist=0.9999999999999998  rot\_y=0.0  Orientation=**(0,180.0,0)** |

There are two errors with this output – firstly, the vertexes are not in the correct order. It should be anti-clockwise starting from the bottom left, but instead are clockwise starting from the bottom right. Secondly, the orientation is 180 degrees out. This is consistent with the first error, though incorrect nonetheless.

Having further examined the code that computes the orientation, I also noticed that I made an invalid assumption about marker faces being vertical, so the x and z rotations always returns as 0. This will give the wrong orientation values for the top face. These issues are discussed further in the evaluation.

## Testing Other Requirements

By manipulating the configuration files, it is fairly easy to set the robots into particular unusual situations to test or verify the behaviour. Some examples are given below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| # | Test | Purpose | Expected Outcome | Actual Outcome |
| 1 | Run the simulation with a robot that drives sets both its motors to 100, then after 5 seconds, sets them to -100. | To test that robot programs can change their motor speeds to drive their robot forwards and backwards.  Requirement 7b. | The robot moves forwards at a roughly constant speed for 5 seconds, then stops and moves backwards until it hits the wall. | **As expected.** |
| 2 | Run the simulation with a robot that drives sets one of its motors to 100, then after 5 seconds, sets it to -100. | To test that robots will correctly turn when the motors have different set powers.  Requirement 7c. | The robot will turn clockwise at a roughly constant speed for 5 seconds, and then stop and turn anti-clockwise until the simulation ends. | **As expected.** |
| 3 | End the simulation with a token of each colour in the yellow team’s zone, and the red robot touching the red token. | To test that scoring is calculated correctly, robots are awarded a point for leaving their zone, and tokens score with the correct priority.  Requirement 2b. | The yellow team scores 8 points.  The red team scores 4 points. | **As expected.** |

### Sample of Test Results

**Test 3:**

|  |  |
| --- | --- |
|  | In Controller: Simulation no longer running. Calculating scores.  Team 0 scored 8 point(s)!  Team 1 scored 0 point(s)!  Team 2 scored 4 point(s)!  Team 3 scored 0 point(s)! |

## System Testing

The purpose of system testing is to ensure that all features of the program work in conjunction with each other. At this level, the expected outcome is more vague than for unit tests or component tests, and is sometimes just “The simulation runs to completion, without obvious errors”.

These tests inevitably exercise the scoring, display, physics, motors, and vision. The noise parameters mean that the runs will vary even if the runs and initial test conditions are the same.

|  |  |  |  |
| --- | --- | --- | --- |
| # | Test | Purpose | Actual Outcome |
| 1 | Run the simulation without any robots. | To test that the program does not break if not given any robot arguments. | **As expected.** |
| 2 | Run the simulation with 4 robots that do nothing except repeatedly call sleep(). | To test that the simulation can successfully run to completion with the maximum number of robots.  Requirement 2a | **As expected.** |
| 3 | As 2, but press the “close” button on the display window during the simulation. | To test that a simulation can be closed prematurely. | **As expected.** |
| 4 | Run the simulation with a robot that exits immediately after creating a robot. | To test that the simulation will continue if a robot program stops cleanly. | **The simulation enters the main loop, and hangs.** |
| 5 | Run the simulation with a robot that crashes immediately after creating a robot. | To test that the simulation will continue if a robot program stops due to an exception. | **The simulation enters the main loop, and hangs.** |
| 6 | Run the simulation with one robot that sleeps through the simulation, and another robot that runs a program that moves around and calls see(). | To test that different programs can be run.  Requirement 2c. | **As expected.** |

|  |  |
| --- | --- |
| **Test 1:** | **Test 3:**  In [1]-Thread at 18.0: Yielding control to MainThread  In MainThread at 27.859375: Yielding control to [0]-Thread  In [0]-Thread at 27.859375: Receiving control from MainThread  In [0]-Thread at 27.859375: Exiting ArenaService.waitForOutput()  In Controller: Receiving control from Simulator.  In Controller: Yielding control to Simulator.  In [0]-Thread at 27.859375: Entering ArenaService.waitForOutput()  In [0]-Thread at 27.859375: Exiting ArenaService.waitForOutput()  In Controller: Receiving control from Simulator.  In Controller: Simulation no longer running. Calculating scores.  Team 0 scored 0 point(s)!  Team 1 scored 0 point(s)!  Team 2 scored 0 point(s)!  Team 3 scored 0 point(s)! |
| **Test 6:** | **Test 5:**  In MainThread at 0: All clients are ready to begin, entering main loop.  In [0]-Thread at 0: Receiving control from MainThread  In Controller: Receiving control from Simulator.  In Controller: Yielding control to Simulator.  In [0]-Thread at 0: Entering ArenaService.waitForOutput()  In [0]-Thread at 0: Yielding control to MainThread  In [1]-Thread at 0: Robot now starting.  In RobotClient: Starting.  In RobotClient: Entering sleep.  In [1]-Thread at 0: Entering RobotService.sleep()  In [1]-Thread at 0: Yielding control to MainThread  In [2]-Thread at 0: Robot now starting.  In RobotClient: Starting.  **Traceback (most recent call last):**  File "TestProgramCrashy.py", line 7, in <module>  print(1 / 0)  ZeroDivisionError: division by zero |

Due to the design flaw already mentioned in the Multi-Threading and Processing section of my design section, the simulation cannot handle gracefully a robot program crashing or ending during the simulation, and will hang indefinitely. This is discussed further in the evaluation.

## End User Level Testing

For the final stage of testing, I obtained a representative program from one of last year’s Robocon participants. This program was a fairly simple program that might be written during the early stages of development is therefore a good example of what motivated the original request.

I ran four copies of this program in my simulator and have uploaded the video of the simulation to YouTube.

<https://www.youtube.com/watch?v=U-3UNJgwo7E&feature=youtu.be>

# **Evaluation**

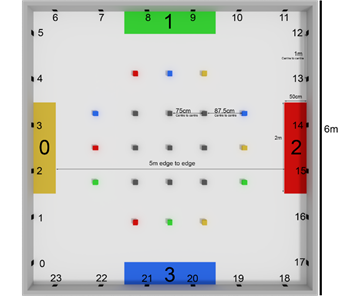
Overall, I feel the project was successful. The simulation is sufficiently complete to be useful for testing simple algorithms without the need for an actual robot. Allowing the user to access the vision and motor functionality of the robot allows them to develop moderately complex programs using only the software. The API, physics and vision are sufficiently close to real life that programs written for the simulator are useful for real robots.

There is room for improvement – ideally, it would be possible to save and replay a simulation. This would mean that if a robot did something unexpected, the moment at which it went wrong could be examined again in closer detail. For this to happen, I would need to further decouple the display code from the simulation itself, to allow some sort of recording and playback system to be inserted between the two. The design does already isolate the display from the physics, so this would be a fairly large but also localised change.  
Sadly though, there was not enough time available to complete this feature, so it was not started.

Additionally, the user interface is currently rather poor. While the intended users are programmers and therefore technically inclined, the current system of starting the simulator using a Terminal command is not user friendly and users would need to be taught how to do it.

## Requirements Review

1. I will build a simple robot simulator, capable of being given up to four python programs for a robot and running them in a virtual environment to mimic the Hills Road Robocon competition.  
   See extended response above.
2. Simulations must be orchestrated in order to allow for:
   1. Multiple robot programs must be able to connect to the same simulated arena. Programs must be started together and stopped after a predetermined time expires.
   2. When the simulation time expires, the scores of the competition must be accessible.  
      The scores of the competition are printed to the console output at the end of the simulation. The simulation display remains in a frozen “limbo” state for 10 seconds to allow the user to examine the final state of the arena.
   3. Multiple robots must be able to use different programs and different configuration settings.
   4. [Ext] The entire process must be repeatable for automated testing.  
      This requirement was not met, as it had no real practical application until better recording and replay facilities were available, and those were not implemented for the reasons explained in the extended response above.
3. It must be possible to view the simulation taking place:
   1. Must display simulation in real time –the positions and orientations of all objects must be updated as they change in the simulation.
   2. Objects associated with a team (gold, zones, robots) must be coloured according to the team they belong to.
   3. When the vision function is called, a triangle representing the field of view of the robot must be drawn. Visible objects will be highlighted.  
      For time reasons, this was only partially implemented. Visible objects are highlighted, though the field of view triangle is not drawn.
   4. Must display the standard output of the robots in real time – all print statements must be displayed to the user as they’re sent.  
      Print statements from the robot processes are printed to the console output 8 times a second.
4. The simulator must work with unmodified code, though a small number of exceptions are allowable:
   1. Robot servos and GPIO collections will exist like in an actual robot, but will be empty, so programs that attempt to use them will not work.
   2. A small number of functions (such as *sleep* or *print*) will need to be changed. These changes must be possible using only simple Find and Replace tools.
      1. [Ext] The syntax changes are made automatically by the simulator, so the simulator can input completely unmodified code.  
         For time reasons, this was not implemented.
5. The simulated arena must be set up as per the rules of the competition:
   1. Immovable and impassable walls must constrain the arena to a 6x6 meter square.
      1. They have 24 markers positioned 0.5m above the ground – each wall has 6 markers, with their centres spaced 1m apart, starting 0.5m from the corner.
      2. The markers each have an Id number, which begins at 0 at the bottom of the leftmost wall, and increases clockwise.
   2. Zones of dimensions 0.5m by 2m are positioned against the centre of each wall, and each correspond to a team.
   3. 25 tokens populate the Arena:
      1. Tokens are 0.11m cubes, with a 0.1m square marker centred on each face. This marker displays the Id of the token.
      2. “Ore” tokens have unique Ids ranging from 32 to 41. Each team has their own “Gold” tokens, which have unique Ids ranging from 42 to 44 for team 0, 45 to 47 for team 1, etc.
      3. “Gold” tokens must begin the game placed 1.375 meters from the arena walls – each team has one “Gold” token in front of every other team’s zone, positioned as in the diagram. 0.875m inwards from that, the “Ore” tokens must be placed 0.75m apart from each other.
      4. [Ext] The positions of the tokens can be configured using a configuration file, to improve forwards compatibility with future competitions.



1. 2D physics should be calculated in the arena to simulate an actual competition. This is done almost entirely using pygame’s inbuilt functions, though there are some specifications that I need to implement:
   1. Different objects in the arena have different masses:
      1. Walls are immovable.
      2. Tokens are (almost) weightless compared to robots.
      3. Robots can have varying weights.
   2. If an object is not undergoing any forces, it should stop almost immediately, to simulate the high friction carpet.  
      This is achieved using a “force damping” factor of almost 1 – this creates an effect similar to a sticky goo covering the arena, which (at high levels) is very close to the desired carpet effect.
2. Changing the motors property of the robot in the test program must drive the robot in the simulation.
   1. The robot has two motors, with powers ranging from -100 to 100.
   2. Every physics step, the robot will move forwards (or backwards) depending on the powers set in the test program.
   3. The simulation should apply the correct rotational moment if the two motors have a different power.  
      This (and 7b) is achieved by applying forces to two points on the robot to emulate motors attached to wheels. The moment will then be resolved in the next physics step.
3. Calling the see() function in the test program must return a list of markers visible to the robot in the simulation.
   1. A marker is visible if:
      1. All four corners are entirely unobstructed.
      2. All four corners are within the field of view.  
         It is worth noting that for the purposes of making these calculations easier, the “image” is approximated to a circle instead of a rectangle. This doesn’t affect the final result much, though it is inaccurate.
      3. The sides of the resultant parallelogram projection onto the camera are longer than the configured threshold for the viewing robot.
   2. Markers must be returned in the format specified in the documentation (<https://hr-robocon.org/docs/vision.html>). In addition to following the object structure, I will need to return:
      1. Information about the marker, including:
         1. The Id of the marker.
         2. The type of the marker (wall or token).
         3. The type of token (none, ore, gold, or fools gold). This is from the perspective of the robot, so if the robot is on team 2, team 2’s gold is returned as gold.
         4. The size of the marker in metres.
         5. The resolution of the image that the marker was taken in.
         6. The time at which the image was taken.
      2. For every corner of the marker, and its centre, I need to return:
         1. Its position in a Cartesian coordinate system relative to the camera.
         2. Its position in a Polar coordinate system relative to the camera.
         3. The position in the raw image that the camera took.
      3. The orientation of the marker (relative to the camera) must be returned.  
         This feature has a bug and does not return the correct value.
   3. If an object is moving, any markers on it cannot be seen (though it can still obstruct other markers). If a camera is moving, it will return no markers. These restrictions do not apply if motion blur is disabled for that robot in its configuration options.
4. The user must be able to change a configuration file (per team) to customise several parameters regarding that team’s robot:  
   This is implemented as a json file for each team, which can be edited to change the parameters for the robot on that team.
   1. Starting position of robot (relative to their zone).
   2. Width, Length and Height of robot.
   3. Mass of robot.
   4. Distance between the wheels (axle length) of robot.
   5. Height of camera above the ground.
   6. Base speed of motors.
   7. The Field of View of the camera.
   8. The extent of simulated “hardware noise”:
      1. Motor offset range (maximum possible difference between the motors).
      2. Camera noise – necessary pixel threshold, with random variance factor.
   9. Toggling motion blur.
5. [Ext] It should be possible to save and replay a simulation:  
   This requirement was not implemented due to time constraints – while it would have been a useful tool, it was not practical to complete it in the time available.
   1. At every timestep, enough information about the simulation must be recorded to allow to be “reconstructed” for the display:
      1. The positions and orientations of all movable objects.
      2. The times when a robot called print(), and what it output.
      3. The times when a robot called its vision function, and the objects it saw.
   2. Must be able to playback a competition from a file.
   3. Must be able to pause, speed up, slow down or jump in the “timeline” of the playback.

## Known Issues

### Client Programs Not Returning

As noted in the design document, the simulation requires well behaved clients. If a client crashes, ends or otherwise never calls a function that causes it to sleep, the simulation cannot recover and hangs indefinitely. This could be partially addressed by changing the RobotClient library to use a Context Manager class. Then if the client program terminates, the Context Manager can inform the simulation of an early exit, and the simulation can continue without that client.

This does not help the case of a client program looping forever, but there are no simple solutions for that problem.

### Orientations of Markers Incorrectly Returning

As noted in the testing document, the orientations of the markers returned by the see() function are wrong. The same process that found the error could be used to isolate the root cause – setting up a carefully constructed arrangement and using the trace function to display intermediate calculations which can then be checked by hand. This is laborious but will almost certainly converge on the error.

However, the distance and the angle to the centre point of the marker are correct, and these are the only two values typically used to steer the robot. This means that the errors are unlikely to affect most client programs, and also explains why the error was discovered too late to attempt the fix as suggested above.

## Client Feedback

Overall, my client was happy with the software. I demonstrated the program by running a simulation of four robots, all of them running a program written by a participant of last year’s Robocon (the same program used in my End User Level Testing). As the simulation ran, I explained various aspects of the simulation and showed him how to access the configuration files.  
My client was satisfied with the accuracy of the physics, the quality of the display and base functionality (multiple robots, vision, and configuration).

There were a few areas that the client requested improvements on:

* The client was not satisfied with the current method of starting the simulator (using a Terminal command) and wanted a more user friendly way to begin a simulation.
* Similarly, the client wanted an easier way to view the scores at the end of the competition, so a terminal window would not be necessary.
* The client was mostly happy with the API compatibility, but requested that it be possible for a program to be entered completely unmodified (assuming it doesn’t interface with the GPIO or servos).
* The client wanted it to be easier to identify which way a robot was pointing, as it’s impossible to distinguish between robots facing in opposite directions.

While I didn’t have time to solve these issues after the meeting, the issues could be rectified as follows:

* To make it easier to begin a simulation, I could write a short program that prompts the user for up to 4 robot program files, then constructs and executes the terminal command to start the simulator with those programs.
* This same program could print the scores of the competition at the end.
* To allow a program to be entered completely unmodified, I could write a function to read the text content of a python file, process it with regular expressions to carry out the necessary “find and replace” operations, and then save the new python file to be executed in place of the one provided. This new file would then be deleted at the end of the simulation.
* An arrow sprite could be added to the robot to indicate the direction it was facing.

None of these changes would require a significant change in the design, though weren’t implemented for time reasons.