# Configuration Data Files

## Robot 0 Config.json

[{

"Distance Between Wheels" : 0.4,

"Maximum Motor Power" : 1.0,

"Motor Noise Range" : 0.1,

"Starting Position" : [0.0, 0.0],

"Width" : 0.3,

"Length" : 0.4,

"Height" : 0.3,

"Mass" : 1.0,

"Camera Height" : 0.3,

"Camera Field of View" : 45.0,

"Marker Pixels Minimum" : 40,

"Marker Pixels Noise Range" : 5,

"Ignore Motion Blur" : false

}]

## Token Position Config.json

1. {

"Ore" :

[

[-0.75, -0.75],

[-0.75, 0.0],

[-0.75, 0.75],

[0.0, -0.75],

[0.0, 0.0],

[0.0, 0.75],

[0.75, -0.75],

[0.75, 0.0],

[0.75, 0.75]

],

"Team 0 Gold" :

[

[0.75, -1.625],

[0.75, 1.625],

[1.625, 0.0]

],

"Team 1 Gold" :

[

[-1.625, -0.75],

[0.0, -1.625],

[1.625, -0.75]

],

"Team 2 Gold" :

[

[-1.625, 0.0],

[-0.75, -1.625],

[-0.75, 1.625]

],

"Team 3 Gold" :

[

[-1.625, 0.75],

[0.0, 1.625],

[1.625, 0.75]

]

}

# **Source Code**

## Controller.py

1. **import** sys

**import** argparse

**import** subprocess

**import** time

**import** xmlrpc.client

**def** trace(text):

*""" Print a message to Standard Error, to avoid polluting the Standard Output (which is read by some processes)."""*

print("In Controller: " + text, file = sys.stderr)

**if** \_\_name\_\_ == "\_\_main\_\_":

*"""Main program."""*

*#Construct a list of all programs to test.*

parser = argparse.ArgumentParser("RobotClient")

parser.add\_argument("--test", action="append", help="The filename of the program to test.")

arguments = parser.parse\_args()

programsToTest = []

**if** arguments.test:

**for** testProgram **in** arguments.test:

programsToTest.append(testProgram)

*#Trim the list to contain a maximum of 4 robot programs.*

**del** programsToTest[4:]

*#Start the simulator as a subprocess, then wait for it to print the URL of the ArenaThread's xmlrpc server.*

*#Once the controller recieves the URL, it connects to the server and stores the connection.*

trace("Starting simulator process.")

simulator = subprocess.Popen( ["python3", "Simulator.py"], stdout=subprocess.PIPE)

trace("Started simulator, waiting for URL.")

arena = **None**

**for** line **in** simulator.stdout:

text = line.decode('UTF-8').rstrip()

*#Printed lines come in as a Bytes object, and must be converted to a string.*

**if** text[0:12] == "Arena URL = ":

trace("URL recieved, connecting to the arena service.")

arena = xmlrpc.client.ServerProxy(text[12:])

trace("Connected to arena service.")

**break**

*#Starts the robot programs as subprocesses.*

robots = []

teamNumber = 0

**for** testProgram **in** programsToTest:

trace("Creating robot number " + str(teamNumber))

serviceURL = arena.createRobot(teamNumber)

trace("Robot created, starting test program subprocess.")

robots.append( subprocess.Popen([ "python3", testProgram, "--url", serviceURL ]) )

trace("Test program subprocess created.")

teamNumber += 1

trace("All robots created, waiting for start.")

arena.waitForStart()

trace("Receiving control from Simulator.")

*#Main loop.*

isSimulatorRunning = **True**

**while** isSimulatorRunning:

trace("Yielding control to Simulator.")

output = arena.waitForOutput(30)

trace("Receiving control from Simulator.")

isSimulatorRunning = output[0]

messages = output[1]

**for** message **in** messages:

print(message)

trace("Simulation no longer running. Calculating scores.")

*#At this stage, the simulation has finished, and the simulator is waiting for a "terminate" call*

*#once the arena thread has finished.*

scores = arena.getScores()

teamNumber = 0

**for** score **in** scores:

print("Team " + str(teamNumber) + " scored " + str(score) + " point(s)!")

teamNumber = teamNumber + 1

time.sleep(10)

*#Allow 10 seconds for the user to view the final state of the simulation.*

trace("Scores calculated, yielding control to Simulator.")

arena.terminate()

trace("Receiving control from Simulator.")

*#Shutdown all subprocesses cleanly.*

arena = **None**

trace("Waiting for Simulator finish.")

simulator.wait()

trace("Simulator has finished.")

**for** robot **in** robots:

trace("Waiting for robot test program to finish.")

robot.wait()

trace("Robot test program has finished.")

trace("All subprocesses have finished. Simulation successful.")

## SimBase.py

1. **import** sys

**import** threading

**import** xmlrpc.server

**import** pymunk

**import** json

**import** math

**import** random

*"""Global Variables"""*

*#The pymunk "world". This is initialised when the ArenaThread is created, but stored here for visibility.*

space = **None**

*#The current time of the simulation (in seconds).*

theTime = 0

*#The time at which the simulation ends (in seconds).*

endTime = 180

*#A threading event used to block the main (Simulator) thread.*

mainGate = threading.Event()

*#A list of all running rpcThreads.*

rpcThreads = []

*#A list of all the print statements for the controller to print in the next timestep.*

pendingOutput = []

*#Lists containing all the bodies of the respective type that are currently in the arena.*

wallSegments = []

tokens = []

robots = []

zones = []

*"""Global Helper Functions"""*

**def** trace(text):

*"""Print a message to Standard Error, to avoid polluting the Standard Output (which is read by some processes)."""*

id = threading.current\_thread().getName()

line = "In " + id + " at " + str(theTime) + ": " + str(text)

print(line, file = sys.stderr)

**def** isSimulationRunning():

*"""Returns if the simulation has finished running."""*

**return** theTime < endTime

**def** sanitiseInput(input, datatype, default, minimum = **None**, maximum = **None**):

*"""Takes an input and ensures that it is the correct datatype, and that it is within the allowable range.*

*If it is not, the value will be set to an allowable value, to allow the simulation to run."""*

**assert**( (minimum == **None**) **or** (maximum == **None**) **or** (minimum < maximum) )

**assert**( (minimum == **None**) **or** (minimum <= default) )

**assert**( (maximum == **None**) **or** (maximum >= default) )

**if** isinstance(input, datatype):

**if** minimum != **None**:

**if** input < minimum:

**return** minimum

**if** maximum != **None**:

**if** input > maximum:

**return** maximum

**return** input

**else**:

**return** default

*"""Threading"""*

**class** RpcThread(threading.Thread):

*"""A base class for the ArenaThread and RobotThread classes*

*this contains all the common functions for blocking, unblocking and stopping a thread."""*

**def** \_\_init\_\_(self):

*"""Initialises the thread, creating common attributes to all threads.*

*The server is created differently depending on the type of thread, so is set to None for now."""*

super().\_\_init\_\_(daemon = **True**, \

name="[{}]-Thread".format(len(rpcThreads)))

*#daemon makes the program run more "in the background", and the thread is named*

*#based on when it was added (first thread is "[0] thread", etc)*

self.wakeUpTime = 0

self.gate = threading.Event()

self.server = **None**

self.isReadyToStart = **False**

**def** block(self):

*"""Block the thread, and unblocks the main thread."""*

self.gate.clear()

trace("Yielding control to MainThread")

mainGate.set()

self.gate.wait()

trace("Receiving control from MainThread")

**def** unblock(self):

*"""Unblocks the thread, blocking the main thread (unless the simulation has ended)."""*

mainGate.clear()

self.gate.set()

mainGate.wait()

**def** shutdownAndWaitToExit(self):

*"""Exits the thread cleanly, yielding the program until the shutdown is complete."""*

trace("Releasing " + self.name + " to shut down.")

self.gate.set()

self.server.shutdown()

self.join()

trace(self.name + " has shut down.")

*#For some reason, the function for "cleanly terminate thread" is "join".*

*"""Pymunk Body Classes"""*

*#The rotation corresponding to each team.*

*#For example, relative to an object created for team 0, an object created for team 1 is*

*# rotated -90 degrees about the origin.*

\_teamAngles = {

0 : 0,

1 : math.pi / -2,

2 : math.pi,

3 : math.pi / 2

}

**class** Robot(pymunk.Body):

*"""This class is derived from the body class, and in addition to the base pymunk body attributes.*

*it contains all the information regarding that robot's configuration options,*

*and some flags on if the robot has left it's zone or is moving.*

*It initialises itself using the config file accociated with it's team number."""*

**def** \_\_init\_\_(self, teamNumber):

*"""Initialises the body using information from the config file accociated with that teamNumber."""*

super().\_\_init\_\_(body\_type = pymunk.Body.DYNAMIC)

self.teamNumber = teamNumber

**with** open("Robot " + str(teamNumber) + " Config.json") **as** RobotConfig:

InitialiseDictionary = json.loads(RobotConfig.read())[0]

*#Create the pymunk body and shape:*

*#pymunk shapes cannot have a width or length of 0, so the minimum footprint is 1cm by 1cm.*

self.width = sanitiseInput(InitialiseDictionary["Width"], float, 0.3, 0.01, 0.4)

self.length = sanitiseInput(InitialiseDictionary["Length"], float, 0.4, 0.01, 0.4)

self.height = sanitiseInput(InitialiseDictionary["Height"], float, 0.4, 0)

*#These constants are useful when constructing the object:*

halfWidth = self.width / 2

halfLength = self.length / 2

startingOffset = pymunk.Vec2d(

sanitiseInput(InitialiseDictionary["Starting Position"][0], float, 0, -0.25 + halfLength, 5.75 - halfLength),

sanitiseInput(InitialiseDictionary["Starting Position"][1], float, 0, -3 + halfWidth, 3 - halfWidth)

)

self.position = (pymunk.Vec2d(-2.75, 0) + startingOffset).rotated(\_teamAngles[teamNumber])

self.angle = \_teamAngles[teamNumber]

points = [(-halfLength, -halfWidth), (halfLength, -halfWidth), (halfLength, halfWidth), (-halfLength, halfWidth)]

box = pymunk.Poly(self, points)

*#The mass can't be set until the shape is constructed, so this is stored as a variable for later.*

*#It also can't be 0, so the minimum weight is 1 gram.*

box.mass = sanitiseInput(InitialiseDictionary["Mass"], float, 1, 0.001)

*#For whatever reason, collision\_types have to be integers.*

*#I've decided that collision\_type 1 is for robots, and 2 is for tokens.*

box.collision\_type = 1

box.elasticity = 0

box.friction = 0.5

space.add(self, box)

*#Initialise robot specific values:*

self.\_axleLength = sanitiseInput(InitialiseDictionary["Distance Between Wheels"], float, 0, 0)

baseMaxPower = sanitiseInput(InitialiseDictionary["Maximum Motor Power"], float, 1, 0)

powerOffset = sanitiseInput(InitialiseDictionary["Motor Noise Range"], float, 0, 0)

random.seed()

self.\_leftMaxPower = baseMaxPower + random.uniform(0, powerOffset / 2)

self.\_rightMaxPower = baseMaxPower + random.uniform(0, powerOffset / 2)

*#These will be set when the service recieves a call to update the motor power.*

self.leftPower = 0

self.rightPower = 0

*#Initialise camera values:*

self.cameraHeight = sanitiseInput(InitialiseDictionary["Camera Height"], float, 0.3, 0)

*#The field of view is input as the full angle between opposite sides in degrees, but used as the half angle in radians.*

*#To make this conversion, the input is multiplied by math.pi/360.*

self.fieldOfView = sanitiseInput(InitialiseDictionary["Camera Field of View"], float, 45, 0, 360) \* math.pi / 360

self.markerPixelsMinimum = sanitiseInput(InitialiseDictionary["Marker Pixels Minimum"], int, 0, 0)

self.markerPixelsNoise = sanitiseInput(InitialiseDictionary["Marker Pixels Noise Range"], int, 0, 0)

self.isIgnoringMotionBlur = sanitiseInput(InitialiseDictionary["Ignore Motion Blur"], bool, **False**)

self.hasLeftZone = **False**

robots.append(self)

@property

**def** isMoving(self):

*"""Returns if the object is moving.*

*Small thresholds are acceptable, as objects in pymunk usually take a while to stop moving entirely."""*

**return** self.velocity.length > 0.02 **or** self.angular\_velocity > 0.05

**def** applyMotorForce(self):

*"""Applies the motor forces to the robot body."""*

leftMotorPower = ( self.leftPower / 100 ) \* self.\_leftMaxPower

self.apply\_force\_at\_local\_point( (leftMotorPower, 0), (0, self.\_axleLength/2) )

rightMotorPower = ( self.rightPower / 100 ) \* self.\_rightMaxPower

self.apply\_force\_at\_local\_point( (rightMotorPower, 0), (0, self.\_axleLength/-2) )

**def** checkIfLeftZone(self):

*"""Checks if the robot is outside its zone, and updates the flag if it is."""*

zone = zones[self.teamNumber]

zoneBB = **None**

*#zone.shapes is a set that (in my simulation) only ever contains one item*

*#but since it is a set it must be iterated through with a for loop*

**for** zoneBox **in** zone.shapes:

zoneBB = zoneBox.cache\_bb()

**for** shape **in** space.shapes:

body = shape.body

**if** body == self:

*#if the shape has entirely left the zone*

**for** robotShape **in** body.shapes:

**if** **not**( zoneBB.contains(robotShape.cache\_bb()) ):

self.hasLeftZone = **True**

**class** WallSegment(pymunk.Body):

*"""This class is derived from the body class, and in addition to the base pymunk body attributes, it contains*

*the id of the wall segment and a list of timestamps the four robots last saw it at."""*

**def** \_\_init\_\_(self, segmentId, offset, teamSide):

super().\_\_init\_\_(body\_type = pymunk.Body.STATIC)

rotation = \_teamAngles[teamSide]

self.position = pymunk.Vec2d(-3, offset - 2.5 )

self.position = self.position.rotated(rotation)

self.angle = rotation

*#Centred on the middle of the side pointing towards the arena.*

halfLength = 0.5

width = 0.1

points = [(-width, -halfLength), (-width, halfLength), (0, halfLength), (0, -halfLength)]

box = pymunk.Poly(self, points)

space.add(self, box)

wallSegments.append(self)

self.id = segmentId

self.lastSeenList = [-5, -5, -5, -5]

**class** Zone(pymunk.Body):

*"""This class is derived from the body class, and in addition to the base pymunk body attributes, it contains*

*the team accociated with it, and a function to return a list of all the tokens fully within it's bounds."""*

**def** \_\_init\_\_(self, teamNumber):

super().\_\_init\_\_(body\_type = pymunk.Body.STATIC)

rotation = \_teamAngles[teamNumber]

self.teamNumber = teamNumber

*#Half of the sensor zone exists inside the wall.*

*#This is to ensure tokens stay in the sensor even when pushed up against the wall.*

self.position = pymunk.Vec2d(-3, 0)

self.position = self.position.rotated(rotation)

self.angle = rotation

halfWidth = 1

halfLength = 0.5

points = [(-halfLength, -halfWidth), (halfLength, -halfWidth), (halfLength, halfWidth), (-halfLength, halfWidth)]

box = pymunk.Poly(self, points)

box.sensor = **True**

space.add(self, box)

zones.append(self)

**def** getTokensInZone(self):

*"""Returns a list of all tokens fully contained within the zone (as opposed to just touching)."""*

validTokens = []

**for** shape **in** self.shapes:

zoneBB = shape.cache\_bb()

**for** shapeInfo **in** space.shape\_query(shape):

body = shapeInfo.shape.body

**if** isinstance(body, Token):

*#must be entirely contained within zone*

**for** tokenShape **in** body.shapes:

**if** zoneBB.contains(tokenShape.cache\_bb()):

validTokens.append(body)

**return** validTokens

**class** Token(pymunk.Body):

*"""This class is derived from the body class, and in addition to the base pymunk body attributes, it contains*

*the id of the token, corresponding type, and a list of timestamps the four robots last saw it at. It also contains*

*a function which returns who and what the token is currently scoring for.*

*"""*

**def** \_\_init\_\_(self, TokenId, TokenType, XPosition, YPosition):

super().\_\_init\_\_(body\_type = pymunk.Body.DYNAMIC)

*#radius is a useful constant for construction purposes*

*#it represents the distance from the centre of the box to an edge*

radius = 0.055

self.position = (XPosition, YPosition)

points = [(-radius, -radius), (radius, -radius), (radius, radius), (-radius, radius)]

box = pymunk.Poly(self, points)

box.collision\_type = 2

*#For whatever reason, collision\_types have to be integers.*

*#I've decided that collision\_type 1 is for robots, and 2 is for tokens.*

box.mass = 0.02

box.elasticity = 0

box.friction = 0.5

space.add(self, box)

tokens.append(self)

self.id = TokenId

self.type = TokenType

self.lastSeenList = [-5, -5, -5, -5]

**def** getScore(self, robotCollisions):

*"""Takes a list of all current collisions between robots and tokens, and returns a tuple containing the number of*

*points the token is worth, and the team that those points are being scored for. If the token is not scoring for anyone,*

*it scores 0 points for team 0.*

*As per the rules, tokens score for one team only, and for the highest (absolute) score they are valid for.*

*If two robots are touching a token, neither team can score points for "controlling" it."""*

*#List of tuples of the score value, and the team that it would be awarded to.*

potentialScores = []

**for** listOfCollisionsForARobot **in** robotCollisions:

robotNumber = robotCollisions.index(listOfCollisionsForARobot)

**if** self.id **in** listOfCollisionsForARobot:

**if** self.type == "Ore":

potentialScores.append( (1, robotNumber) )

**elif** self.type == "Team " + str(robotCollisions.index(listOfCollisionsForARobot)) + " Gold":

potentialScores.append( (3, robotNumber) )

**else**:

*#Must be a different team's gold.*

potentialScores.append( (-1, robotNumber) )

**if** len(potentialScores) > 1:

*#Multiple robots touching token, all "controlling" scores invalid.*

potentialScores = []

**for** zone **in** zones:

**if** self **in** zone.getTokensInZone():

**if** self.type == "Ore":

potentialScores.append( (5, zone.teamNumber) )

**elif** self.type == "Team " + str(zone.teamNumber) + " Gold":

potentialScores.append( (7, zone.teamNumber) )

**else**:

*#Must be a different team's gold.*

potentialScores.append( (-2, zone.teamNumber) )

highestScore = (0, 0)

**for** potentialScore **in** potentialScores:

**if** abs(potentialScore[0]) > abs(highestScore[0]):

highestScore = potentialScore

**return** highestScore

@property

**def** isMoving(self):

*"""Returns if the object is moving.*

*Small thresholds are acceptable, as objects in pymunk usually take a while to stop moving entirely."""*

**return** self.velocity.length > 0.02 **or** self.angular\_velocity > 0.05

## Simulator.py

1. **import** threading

**import** time

**import** random

*#my modules*

**import** SimBase

**import** SimArena

**import** SimDisplay

**if** \_\_name\_\_ == "\_\_main\_\_":

*"""Main program."""*

SimBase.trace("Simulator starting.")

*#Create threads for all participants.*

SimBase.mainGate.clear()

SimBase.trace("Creating ArenaThread.")

SimBase.rpcThreads.append( SimArena.ArenaThread() )

SimBase.trace("Starting ArenaThread.")

SimBase.rpcThreads[0].start()

*#The robot rpcThreads are created by the ArenaThread. When it finishes, it'll unblock the mainGate.*

SimBase.trace("Simulator is waiting for clients to be ready to begin.")

SimBase.mainGate.wait()

SimBase.trace("All clients are ready to begin, entering main loop.")

*#Create the display, and enter the main loop.*

display = SimDisplay.Display()

**while** SimBase.isSimulationRunning():

**for** thread **in** SimBase.rpcThreads:

**if** SimBase.theTime >= thread.wakeUpTime:

thread.unblock()

**for** robot **in** SimBase.robots:

*#Apply the motor forces for this step and check if the robot has left its zone.*

robot.applyMotorForce()

**if** **not** robot.hasLeftZone:

robot.checkIfLeftZone()

SimBase.theTime += 1/64

SimBase.space.step(1/64)

display.updateDisplay()

display.processInputs()

*#Exit main loop.*

*#Unblock the ArenaThread to allow it to run post-simulation functions (namely calculating the score).*

SimBase.trace("Yielding control to [0]-Thread")

SimBase.rpcThreads[0].unblock()

*#Once the main thread is unblocked, the Simulator and all the robot threads can shutdown.*

**for** thread **in** SimBase.rpcThreads:

thread.shutdownAndWaitToExit()

SimBase.trace("Simulator process ends")

## SimArena.py

1. **import** sys

**import** threading

**import** xmlrpc.server

**import** time

**import** pymunk

**import** json

*#my modules*

**import** SimBase

**import** SimRobot

**def** \_tokenTypeToInteger(tokenType):

*"""A helper function to convert from the type of the token to the first token with that id."""*

typeToInt = {

"Ore" : 32,

"Team 0 Gold" : 42,

"Team 1 Gold" : 45,

"Team 2 Gold" : 48,

"Team 3 Gold" : 51

}

**return** typeToInt[tokenType]

**class** ArenaService:

*"""Handles the creation of the arena and provides services to the Controller."""*

**def** \_\_init\_\_(self):

*"""Initialises the arena, creating and configuring the space, creating a collision handler to track*

*all active token and robot collisions, and populates the arena with walls, zones and tokens (but not robots)."""*

SimBase.space = pymunk.Space()

SimBase.space.damping = 0.01

*#Set up collision handler for when a robot touches a token.*

self.\_scoringCollisions = []

*#This is a list, indexed by the id of the robot, and the value is a list of token ids that are colliding with that robot.*

*#Since the arena doesn't know how many robots are going to be in the simulation yet, the list is populated with*

*#empty lists as robots are added.*

collisionHandler = SimBase.space.add\_collision\_handler(1, 2)

collisionHandler.begin = self.\_robotTokenCollisionBegin

collisionHandler.separate = self.\_robotTokenCollisionEnd

*#Create the walls:*

id = 0

**for** teamSide **in** range(4):

**for** offset **in** range(6):

SimBase.WallSegment(id, offset, teamSide)

id += 1

*#Create the zones:*

**for** team **in** range(4):

SimBase.Zone(team)

*#Create the tokens:*

**with** open('Token Position Config.json') **as** TokenConfig:

TokenList = json.loads(TokenConfig.read())

**for** tokenType, tokenPositions **in** TokenList.items():

currentId = \_tokenTypeToInteger(tokenType)

**for** tokenPosition **in** tokenPositions:

*#Sanitise the two coordinates, returning False if it is not the correct datatype,*

*#and restricting the values to within the arena.*

xPosition = SimBase.sanitiseInput(tokenPosition[0], float, **False**, -2.945, 2.945)

yPosition = SimBase.sanitiseInput(tokenPosition[1], float, **False**, -2.945, 2.945)

**if** isinstance(xPosition, float) **and** isinstance(yPosition, float):

SimBase.Token(currentId, tokenType, xPosition, yPosition)

currentId += 1

**def** \_robotTokenCollisionBegin(self, arbiter, space, data):

*"""A function that gets bound to the CollisionHandler,*

*which adds the token/robot pair to the list of active collisions."""*

*#First, work out which of the shapes is the robot, and which is the token.*

token = **None**

robot = **None**

**if** isinstance(arbiter.shapes[0].body, SimBase.Token):

token = arbiter.shapes[0].body

robot = arbiter.shapes[1].body

**else**:

robot = arbiter.shapes[0].body

token = arbiter.shapes[1].body

self.\_scoringCollisions[robot.teamNumber].append(token.id)

**return** **True**

*#This tells the collision handler to handle the physics of the collision normally.*

**def** \_robotTokenCollisionEnd(self, arbiter, space, data):

*"""A function that gets bound to the CollisionHandler,*

*which removes the token/robot pair from the list of active collisions."""*

*#First, work out which of the shapes is the robot, and which is the token.*

token = **None**

robot = **None**

**if** isinstance(arbiter.shapes[0].body, SimBase.Token):

token = arbiter.shapes[0].body

robot = arbiter.shapes[1].body

**else**:

robot = arbiter.shapes[0].body

token = arbiter.shapes[1].body

self.\_scoringCollisions[robot.teamNumber].remove(token.id)

**def** createRobot(self, teamNumber):

*"""Creates a new robot thread (which then creates a robot service and robot body),*

*and returns the connection URL to it's xmlrpc server."""*

**if** **not** SimBase.isSimulationRunning():

**raise** RuntimeError("Attempted to create a robot when the simulation had already ended.")

newThread = SimRobot.RobotThread(teamNumber)

SimBase.rpcThreads.append(newThread)

newThread.start()

url = newThread.getUrl()

self.\_scoringCollisions.append([])

**return** url

**def** getScores(self):

*"""Calculates the scores of each team.*

*This is done by first summing the scores of each token, and then awarding an additional point*

*to the team of each robot that left its zone."""*

scores = [0, 0, 0, 0]

**for** token **in** SimBase.tokens:

score, team = token.getScore(self.\_scoringCollisions)

scores[team] += score

**for** robot **in** SimBase.robots:

**if** robot.hasLeftZone:

scores[robot.teamNumber] += 1

**return** scores

**def** waitForOutput(self, time):

*"""Clears the list of pending messages to send to the Controller, then waits for the simulated*

*time to have elapsed, and then returns a tuple of if the simulation has finished yet, and a list*

*of messages to print to the Standard Output."""*

SimBase.trace("Entering ArenaService.waitForOutput()")

messagesToSend = SimBase.pendingOutput

SimBase.pendingOutput = []

**if** **not** SimBase.isSimulationRunning():

SimBase.trace("Exiting ArenaService.waitForOutput()")

**return** (**False**, messagesToSend)

arenaThread = threading.current\_thread()

arenaThread.wakeUpTime += time

arenaThread.block()

SimBase.trace("Exiting ArenaService.waitForOutput()")

**return** (**True**, messagesToSend)

**def** waitForStart(self):

*"""Marks the arena as ready, then waits until all other threads (the robots) are ready, then blocks itself,*

*unblocking the main thread and allowing it to enter the main simulator loop."""*

SimBase.trace("Entering ArenaService.waitForStart()")

arenaThread = threading.current\_thread()

arenaThread.isReadyToStart = **True**

**for** thread **in** SimBase.rpcThreads:

SimBase.trace("Arena is waiting for " + str(thread.name))

**while** thread.isReadyToStart == **False**:

time.sleep(1)

SimBase.trace("All robots ready, leaving ArenaService.waitForStart()")

arenaThread.block()

**return** **True**

**def** terminate(self):

*"""If called before the simulation has ended, raises an error.*

*Otherwise, blocks the thread and allows the simulator to end the simulation (and terminate this thread)."""*

**if** SimBase.isSimulationRunning():

**raise** RuntimeError("Attempted to terminate the ArenaThread before the simulation had ended.")

arenaThread = threading.current\_thread()

arenaThread.block()

**return** **True**

**class** ArenaThread(SimBase.RpcThread):

*"""A thread that handles the xmlrpc server to communicate with the Controller."""*

**def** run(self):

*"""Initialises the xmlrpc server, then prints connection details to the standard output, where they*

*are caught by the Controller. Then connects the xmlrpc server to the ArenaService, and serves until stopped."""*

self.server = xmlrpc.server.SimpleXMLRPCServer( ('localhost', 0), logRequests=**False** )

address = self.server.server\_address

print("Arena URL = http://{}:{}".format(address[0], address[1]))

sys.stdout.flush() *#flushing the stdout is required to allow the controller to see the message*

self.server.register\_instance(ArenaService())

self.server.serve\_forever()

self.server.server\_close()

## SimRobot.py

1. **import** sys
2. **import** threading
3. **import** xmlrpc.server
4. **import** SimBase
5. **import** SimVision
6. **class** RobotService:
7. *"""Handles the creation of a robot and provides an interface to the simulated robot for the RobotClient.*
8. *Also provides a helper function for the main simulator thread to check if the robot has left its zone.*
9. *Also provides a helper function for the main simulator thread to apply its motor forces."""*
10. **def** \_\_init\_\_(self):
11. *"""Initialises the service, with two motors and the pymunk robot body.*
12. *Since robots are initiated in order, the team number of the robot being created can be*
13. *worked out using the number of existing robots."""*
14. self.leftMotor = 0
15. self.rightMotor = 0
16. teamNumber = len(SimBase.robotServices)
17. self.robotBody = SimBase.Robot(teamNumber)
18. SimBase.robotServices.append(self)
19. **def** getTeamNumber(self):
20. *"""Returns the team number of the robot."""*
21. **return** self.robotBody.teamNumber
22. **def** getMotorPower(self, motorNumber):
23. *"""Returns the power of the motor specified, or 0 if asked for a motor outside the accepted range.*
24. *Throws an exception if the simulation has already finished."""*
25. **if** **not** SimBase.isSimulationRunning():
26. **raise** RuntimeError("Attempted to call a robot function when simulation had already ended.")
27. **if** motorNumber == 1:
28. **return** self.leftMotor
29. **elif** motorNumber == 2:
30. **return** self.rightMotor
31. **else**:
32. **raise** RuntimeError("Invalid motor number given.")
33. **def** setMotorPower(self, motorNumber, newPower):
34. *"""Sets the power of the robot motor specified (capping values at +-100), and returns the power it was set to.*
35. *Throws an exception if the simulation has already finished."""*
36. **if** **not** SimBase.isSimulationRunning():
37. **raise** RuntimeError("Attempted to call a robot function when simulation had already ended.")
38. **if** **not** ( isinstance(newPower, int) **or** isinstance(newPower, float) ):
39. **raise** RuntimeError("Attempted to set motor power to non-numeric value.")
40. **if** newPower < -100:
41. newPower = -100
42. **elif** newPower > 100:
43. newPower = 100
45. **if** motorNumber == 1:
46. self.leftMotor = newPower
47. **elif** motorNumber == 2:
48. self.rightMotor = newPower
49. **else**:
50. **raise** RuntimeError("Attempted to set power of an invalid motor.")
51. **return** newPower
52. **def** print(self, message):
53. *"""Adds a message to the pending output, to be printed by the Controller program when it next recieves them.*
54. *Returns True if successful. Throws an exception if the simulation has already finished."""*
55. **if** **not** SimBase.isSimulationRunning():
56. **raise** RuntimeError("Attempted to call a robot function when simulation had already ended.")
57. SimBase.pendingOutput.append("Robot " + str(self.robotBody.teamNumber) + " at " + str(SimBase.theTime) + " printed: " + message)
58. **return** **True**
60. **def** sleep(self, time):
61. *"""Waits until the simulated time has increased by the specified delay.*
62. *Returns False if the simulation is then no longer running, True otherwise.*
63. *Throws an exception if the simulation has already finished."""*
64. **if** **not** SimBase.isSimulationRunning():
65. **raise** RuntimeError("Attempted to call a robot function when simulation had already ended.")
66. robotThread = threading.current\_thread()
67. robotThread.wakeUpTime += time
68. robotThread.block()
69. **return** SimBase.isSimulationRunning()
70. **def** see(self, res):
71. *"""Calls the see function (providing the robot, resolution and if the image is blurred).*
72. *Returns a dictionary that is used by the RobotClient package to create a list of Marker Objects, or False if the simulation is no longer running.*
73. *Additionally, yields the robot program briefly depending on the resolution given.*
74. *Throws an exception if the simulation has already finished."""*
75. **if** **not** SimBase.isSimulationRunning():
76. **raise** RuntimeError("Attempted to call a robot function when simulation had already ended.")
77. ids = SimVision.see(self.robotBody, res, self.robotBody.isMoving)
78. robotThread = threading.current\_thread()
79. robotThread.wakeUpTime += res[0]\*0.001
80. robotThread.block()
81. **return** ids
82. **def** waitForStart(self):
83. *"""Sets a flag to indicate the robot is ready to start, and blocks itself until the competition begins."""*
84. robotThread = threading.current\_thread()
85. robotThread.isReadyToStart = True
86. robotThread.gate.clear()
87. robotThread.gate.wait()
88. **return** **True**
89. **def** onEachStep(self):
90. *"""Applies the motor forces to the robot body, and if the robot has not left its zone, checks if it has left its zone."""*
91. robot = self.robotBody
92. leftMotorPower = ( self.leftMotor / 100 ) \* robot.leftMaxPower
93. robot.apply\_force\_at\_local\_point( (leftMotorPower, 0), (0, robot.axleLength/2) )
94. rightMotorPower = ( self.rightMotor / 100 ) \* robot.rightMaxPower
95. robot.apply\_force\_at\_local\_point( (rightMotorPower, 0), (0, robot.axleLength/-2) )
96. **if** **not**(robot.hasLeftZone):
97. zone = SimBase.zones[robot.teamNumber]
98. **for** shape **in** zone.shapes:
99. zoneBB = shape.cache\_bb()
100. **for** shapeInfo **in** SimBase.space.shape\_query(shape):
101. body = shapeInfo.shape.body
102. **if** body == robot:
103. #if the shape has entirely left the zone
104. **for** robotShape **in** body.shapes:
105. **if** **not**( zoneBB.contains(robotShape.cache\_bb()) ):
106. robot.hasLeftZone = **True**
107. **return** **True**
108. **class** RobotThread(SimBase.RpcThread):
109. *"""A thread that handles the xmlrpc server to communicate with the robot program under test."""*
111. **def** \_\_init\_\_(self):
112. *"""Creates the xmlrpc server and connects it to the RobotService.*
113. *Unlike ArenaThread, the server is created during \_\_init\_\_ instead of run().*
114. *This allows getURL to be called before the server is started."""*
115. super().\_\_init\_\_()
116. self.server = xmlrpc.server.SimpleXMLRPCServer( ('localhost', 0), logRequests=False )
117. self.server.register\_instance( RobotService() )
118. **def** getUrl(self):
119. *"""Returns the URL of the server."""*
120. address = self.server.server\_address
121. url = "http://{}:{}".format(address[0], address[1])
122. **return** url
124. **def** run(self):
125. *"""Executed when the thread is started. Runs the xmlrpc server until it is stopped when the simulation ends."""*
126. self.server.serve\_forever()
127. self.server.server\_close()

## vector3.py

1. **import** math

**class** Vector3:

*"""An object that represents a vector in 3D space."""*

**def** \_\_init\_\_(self, xComponent, yComponent, zComponent):

self.x = xComponent

self.y = yComponent

self.z = zComponent

*"""Defining these functions overloads their respective python operators,*

*allowing me to use =, +, -, \* and the str() function on Vector3 objects."""*

**def** \_\_str\_\_(self):

*"""Returns a string representation of the vector, allowing it to be print()ed or similar."""*

**return** "Vector3(" + str(self.x) + ", " + str(self.y) + ", " + str(self.z) + ")"

**def** \_\_eq\_\_(self, other):

*"""First, checks if the other object is a Vector3.*

*If they are, it compares the components of the two vectors to see if the vectors are equivalent."""*

**if** isinstance(other, Vector3):

**return** (self.x == other.x) **and** (self.y == other.y) **and** (self.z == other.z)

**else**:

**return** **False**

**def** \_\_add\_\_(self, otherVector3):

*"""Adds the second vector to the first one."""*

**return** Vector3(self.x + otherVector3.x, self.y + otherVector3.y, self.z + otherVector3.z)

**def** \_\_sub\_\_(self, otherVector3):

*"""Subtracts the second vector from the first one."""*

**return** Vector3(self.x - otherVector3.x, self.y - otherVector3.y, self.z - otherVector3.z)

**def** \_\_neg\_\_(self):

*"""Creates a vector multiplied by the scalar constant -1."""*

**return** Vector3(self.x \* -1, self.y \* -1, self.z \* -1)

**def** \_\_mul\_\_(self, constant):

*"""Multiply the vector by a scalar constant:*

*Vector3 \* Constant"""*

**return** Vector3(self.x \* constant, self.y \* constant, self.z \* constant)

**def** \_\_rmul\_\_(self, constant):

*"""Multiply a scalar constant by the vector:*

*Constant \* Vector3"""*

**return** Vector3(self.x \* constant, self.y \* constant, self.z \* constant)

@property

**def** magnitude(self):

*"""Returns the magnitude of the vector."""*

**return** math.sqrt( (self.x \*\* 2) + (self.y \*\* 2) + (self.z \*\* 2) )

@property

**def** unit(self):

*"""Returns the unit vector with the same direction as the vector."""*

vectorMagnitude = self.magnitude

**if** vectorMagnitude == 0:

**raise** ZeroDivisionError("Attempted to get the direction of a null vector.")

**else**:

**return** Vector3( self.x / vectorMagnitude, self.y / vectorMagnitude, self.z / vectorMagnitude )

**def** dot(self, otherVector3):

*"""Calculates the dot product of a vector with another vector."""*

**return** (self.x \* otherVector3.x) + (self.y \* otherVector3.y) + (self.z \* otherVector3.z)

**def** cross(self, otherVector3):

*"""Calculates the cross product of a vector with another vector."""*

**return** Vector3(

(self.y \* otherVector3.z) - (self.z \* otherVector3.y),

(self.z \* otherVector3.x) - (self.x \* otherVector3.z),

(self.x \* otherVector3.y) - (self.y \* otherVector3.x)

)

**def** angleBetween(self, vectorB):

*"""Returns the angle in radians between two vectors."""*

**return** math.acos(self.dot(vectorB) / ( self.magnitude \* vectorB.magnitude ) )

**def** rotateAroundZ(self, angle):

*"""Rotates the vector around the Z axis angle radians.*

*The only rotations used by the function are around the Z axis, so a general function is not needed."""*

**return** Vector3(

self.x \* math.cos(angle) - self.y \* math.sin(angle),

self.x \* math.sin(angle) + self.y \* math.cos(angle),

self.z

)

**def** convertToDictionary(self):

*"""Converts the vector into a dictionary for the purposes of transferring it through XMLrpc."""*

**return** {

"x" : self.x,

"y" : self.y,

"z" : self.z

}

**def** constructFromDictionary(dictionary):

*"""Constructs a vector3 from a dictionary containing values for "x", "y" and "z"."""*

**return** Vector3(dictionary["x"], dictionary["y"], dictionary["z"])

**class** Plane:

*"""An object that represents a bounded plane in 3D space."""*

**def** \_\_init\_\_(self, bottomLeft, bottomRight, topLeft):

*"""Takes three corners, and defines a plane in vector form (point J being the "bottom right" corner, vectors U and V*

*being two vectors on the plane).*

*This also finds the normal to the plane, and finds its cartesian form.*

*"""*

self.\_pointJ = bottomLeft

self.\_vectorU = bottomRight - bottomLeft

self.\_vectorV = topLeft - bottomLeft

@property

**def** pointJ(self):

**return** self.\_pointJ

@property

**def** vectorU(self):

**return** self.\_vectorU

@property

**def** vectorV(self):

**return** self.\_vectorV

@property

**def** normalToPlane(self):

**return** self.\_vectorU.cross(self.\_vectorV)

@property

**def** cartesianA(self):

**return** self.normalToPlane.x

@property

**def** cartesianB(self):

**return** self.normalToPlane.y

@property

**def** cartesianC(self):

**return** self.normalToPlane.z

@property

**def** cartesianD(self):

**return** self.\_pointJ.dot(self.normalToPlane)

**def** isFacingCamera(self, cameraPosition):

*"""The normal to the plane points OUT of the cuboid, due to the order of the corners in the plane.*

*If the vector from pointJ to the camera also points in that same direction, the plane is facing the camera.*

*If the dot product of the two vectors is > 0, they are pointing in the same direction, so the plane is facing the camera.*

*"""*

**return** (cameraPosition - self.\_pointJ).dot(self.normalToPlane) > 0

**def** isObstructingPoint(self, point, cameraPosition):

*"""Takes a point, and the position of the camera.*

*Returns True if the bounded plane obstructs the line between the point and the cameraPosition, False otherwise."""*

*#First, give up if the line is parallel to the plane, to avoid dividing by 0 later.*

direction = point - cameraPosition

**if** self.cartesianA \* direction.x + self.cartesianB \* direction.y + self.cartesianC \* direction.z != 0:

*# direction \* distance = cameraPosition - intersectionPoint*

lamda = (

( self.cartesianD - self.cartesianA\*cameraPosition.x - self.cartesianB\*cameraPosition.y - self.cartesianC\*cameraPosition.z )

/ ( self.cartesianA \* direction.x + self.cartesianB \* direction.y + self.cartesianC \* direction.z)

)

*#if plane of obstruction is between point and camera*

*#The upper bound is 0.999 instead of 1, so that markers don't obstruct themselves.*

**if** lamda > 0 **and** lamda < 0.999:

intersectionPoint = cameraPosition + (direction \* lamda) - self.\_pointJ

mu = intersectionPoint.dot(self.\_vectorU) / (self.\_vectorU.magnitude \*\* 2)

**if** mu > 0 **and** mu < 0.999:

nu = intersectionPoint.dot(self.\_vectorV) / (self.\_vectorV.magnitude \*\* 2)

**if** nu > 0 **and** nu < 0.999:

**return** **True**

**return** **False**

## SimVision.py

1. **import** math

**import** random

**from** vector3 **import** \*

**import** SimBase

**def** \_getVisibleCuboidFaces(body, cameraPosition, height):

*"""Takes a body, the position of the camera, and the height of the body, and returns a list of all faces*

*of the cuboid that are visible to the camera.*

*Faces that do not face the camera are not returned as they cannot be visible, as they are fully obstructed*

*by the faces in that cuboid that face the camera.*

*Additionally, the floor face is not checked, as it can never face the camera."""*

planes = []

*#body.shapes is actually a set, which means it cannot be indexed and I need to iterate through it with a for loop.*

*#Since all my bodies only contain one shape, this for loop will always run exactly once.*

**for** shape **in** body.shapes:

groundVertexes = []

raisedVertexes = []

**for** groundVertex **in** shape.get\_vertices():

x,y = groundVertex.rotated(body.angle) + body.position

groundVertexes.append( Vector3(x, y, 0) )

raisedVertexes.append( Vector3(x, y, height) )

frontLeft = Plane(groundVertexes[0], raisedVertexes[0], groundVertexes[3])

**if** frontLeft.isFacingCamera(cameraPosition):

planes.append(frontLeft)

frontRight = Plane(groundVertexes[1], raisedVertexes[1], groundVertexes[0])

**if** frontRight.isFacingCamera(cameraPosition):

planes.append(frontRight)

backLeft = Plane(groundVertexes[2], groundVertexes[3], raisedVertexes[2])

**if** backLeft.isFacingCamera(cameraPosition):

planes.append(backLeft)

backRight = Plane(groundVertexes[1], groundVertexes[2], raisedVertexes[1])

**if** backRight.isFacingCamera(cameraPosition):

planes.append(backRight)

roof = Plane(raisedVertexes[0], raisedVertexes[1], raisedVertexes[3])

**if** roof.isFacingCamera(cameraPosition):

planes.append(roof)

**return** planes

**def** \_getObstructingPlanesFromBody(obstructingBody, cameraPosition):

*"""Takes a body that could potentially obstruct the camera (a robot or token), and the position of the camera.*

*Returns the three planes that are visible to the camera (and could obstruct it's vision)."""*

obstructionHeight = **None**

**if** isinstance(obstructingBody, SimBase.Robot):

obstructionHeight = obstructingBody.height

**else**:

*#Must be a token.*

obstructionHeight = 0.11

planes = \_getVisibleCuboidFaces(obstructingBody, cameraPosition, obstructionHeight)

**return** planes

**def** \_isMarkerResolvable(markerCornerSet, cameraPosition, FoV, resolution, pixelThreshold):

*"""Takes a set of four corner positions, the position of the camera, the FoV of the camera, the resolution of the image,*

*and the smallest number of pixels that the marker can encompass in the image while still being visible.*

*A marker is resolvable if the angle subtended by the corners of the marker is greater than the minimum resolvable angle*

*(determined by the resolution of the image, field of view of the camera, and the pixelThreshold).*

*If either is too small, the marker would be either shorter or thinner than the pixelThreshold if a picture was taken."""*

*#Avoid divide by 0 errors.*

**if** resolution[0] == 0 **or** FoV == 0:

**return** **False**

pixelsPerRadian = resolution[0] / FoV

minimumResolvableAngle = pixelThreshold / pixelsPerRadian

*#Vectors between three corners and the position of the camera.*

vectorA = markerCornerSet[0] - cameraPosition

vectorB = markerCornerSet[1] - cameraPosition

vectorC = markerCornerSet[3] - cameraPosition

**return** (vectorA.angleBetween(vectorB) > minimumResolvableAngle) **and** (vectorA.angleBetween(vectorC) > minimumResolvableAngle)

**def** \_getMarkerCornersFromWallSegment(body):

*"""Takes a wall segment, and calcuates a list of points in 3D space where the corners of the marker would lie.*

*This is then returned inside another list, for compatibility with tokens (which have multiple markers)."""*

corners = []

markerCentre = Vector3(body.position[0], body.position[1], 0.175)

*#Vector from the centre of the marker to the side.*

markerRadius = Vector3(0, 0.125, 0).rotateAroundZ(body.angle)

corners.append( markerCentre - markerRadius - Vector3(0, 0, 0.125) )

corners.append( markerCentre + markerRadius - Vector3(0, 0, 0.125) )

corners.append( markerCentre + markerRadius + Vector3(0, 0, 0.125) )

corners.append( markerCentre - markerRadius + Vector3(0, 0, 0.125) )

*#return a list of 1 markers*

**return** [corners]

**def** \_getMarkerCornersFromToken(body, cameraPosition):

*"""Takes a token, and returns a list containing three lists of points in 3D space where the corners of*

*the markers visible to the camera would lie."""*

faces = \_getVisibleCuboidFaces(body, cameraPosition, 0.11)

markersCorners = []

**for** face **in** faces:

corners = []

*#accounts for the 5mm border around the markers*

uOffset = face.vectorU\*(5/110)

vOffset = face.vectorV\*(5/110)

corners.append(face.pointJ + uOffset + vOffset)

corners.append(face.pointJ + face.vectorV + uOffset - vOffset)

corners.append(face.pointJ + face.vectorU + face.vectorV - uOffset - vOffset)

corners.append(face.pointJ + face.vectorU - uOffset + vOffset)

markersCorners.append(corners)

**return** markersCorners

**def** \_constructMarkerInfoDictionary(markerCornerSet, body):

*"""Takes a set of corners and the body that the marker is attached to.*

*Returns a dictionary containing all the information needed by the RobotClient module to construct a Marker object."""*

cornerList = []

**for** corner **in** markerCornerSet:

cornerList.append(corner.convertToDictionary())

size = 0

**if** isinstance(body, SimBase.Token):

size = 0.1

**else**:

size = 0.25

**return** {

"Corners" : cornerList,

"Id" : body.id,

"Size" : size

}

**def** see(robot, resolution, isImageBlurred):

*"""Takes the robot attempting to look for markers, the resolution at which the image was taken,*

*and if the image is blurred (caused by the robot moving). Returns a dictionary containing all the*

*information needed by the RobotClient to construct a list of all visible Marker objects.*

*Constructing the Marker objects is done by the RobotClient because it is not possible to send arbitrary*

*object structures using xmlrpc."""*

MarkersList = []

cameraNormal = Vector3( math.cos(robot.angle), math.sin(robot.angle), 0 )

cameraPosition = Vector3( robot.position[0], robot.position[1], robot.cameraHeight ) + ( cameraNormal \* ( robot.length / 2) )

*#Only return any markers if the image is not blurred (or the robot is ignoring blur).*

**if** robot.isIgnoringMotionBlur **or** (**not** isImageBlurred):

random.seed()

*#potentialObstructioningPlanes is a list of planes that could potentially get in the way*

potentialObstructingPlanes = []

*#tokens is a list of tokens, walls is a list of walls*

markedBodies = []

**for** body **in** SimBase.space.bodies:

**if** isinstance(body, SimBase.Robot):

**if** body != robot:

potentialObstructingPlanes.extend(\_getObstructingPlanesFromBody(body, cameraPosition))

**elif** isinstance(body, SimBase.Token):

potentialObstructingPlanes.extend(\_getObstructingPlanesFromBody(body, cameraPosition))

**if** robot.isIgnoringMotionBlur **or** (**not** body.isMoving):

markedBodies.append(body)

**elif** isinstance(body, SimBase.WallSegment):

markedBodies.append(body)

**for** body **in** markedBodies:

**if** isinstance(body, SimBase.Token):

markerCornerSets = \_getMarkerCornersFromToken(body, cameraPosition)

**else**:

markerCornerSets = \_getMarkerCornersFromWallSegment(body)

**for** markerCornerSet **in** markerCornerSets:

*#If the marker is too slanted or too far away for there to be enough pixels to resolve it, skip this marker.*

markerPixelMinimumAdjusted = robot.markerPixelsMinimum + random.randint( -robot.markerPixelsNoise // 2, robot.markerPixelsNoise // 2 )

**if** **not** \_isMarkerResolvable(markerCornerSet, cameraPosition, robot.fieldOfView, resolution, markerPixelMinimumAdjusted):

**continue**

numVisible = 0

**for** markerCorner **in** markerCornerSet:

**if** cameraNormal.angleBetween(markerCorner - cameraPosition) > robot.fieldOfView:

**break**

**for** potentialObstructingPlane **in** potentialObstructingPlanes:

**if** potentialObstructingPlane.isObstructingPoint(markerCorner, cameraPosition):

**break**

**else**:

numVisible += 1

**if** numVisible == 4:

*#Set the time the body was last seen by the looking robot's id, and constructs a dictionary of information*

*#about the vector.*

body.lastSeenList[robot.teamNumber] = SimBase.theTime

MarkersList.append(\_constructMarkerInfoDictionary(markerCornerSet, body))

*#Construct the dictionary needed to build the list of Marker objects on the RobotClient.*

returnDictionary = {

"Resolution" : resolution,

"Field of View" : robot.fieldOfView,

"Camera Position" : cameraPosition.convertToDictionary(),

"Camera Normal" : cameraNormal.convertToDictionary(),

"Timestamp" : SimBase.theTime,

"List of Markers" : MarkersList

}

**return** returnDictionary

## RobotClient.py

1. **import** argparse

**import** sys

**import** xmlrpc.client

**from** vector3 **import** \*

**def** \_trace(text):

*""" Print a message to Standard Error, to avoid polluting the Standard Output (which is read by some processes)."""*

print("In RobotClient: " + text, file = sys.stderr)

MARKER\_ARENA, MARKER\_TOKEN = 'arena', 'token'

TOKEN\_NONE, TOKEN\_ORE, TOKEN\_FOOLS\_GOLD, TOKEN\_GOLD = 'none', 'ore', 'fools-gold', 'gold'

marker\_offsets = {

MARKER\_ARENA: 0,

MARKER\_TOKEN: 32

}

*"""These classes are defined by the robocon documentation.*

*Their names and properties match those described there - they are initialised using a dictionary of information*

*returned by SimVision's see() function."""*

**class** Orientation:

**def** \_\_init\_\_(self, corners, cameraNormal):

cornerVectors = []

**for** corner **in** corners:

cornerVectors.append(constructFromDictionary(corner))

markerPlane = Plane(cornerVectors[0], cornerVectors[1], cornerVectors[3])

markerNormal = markerPlane.normalToPlane

rot\_yRadians = cameraNormal.angleBetween( -markerNormal )

self.rot\_x = 0

self.rot\_y = (rot\_yRadians / math.pi) \* 180

self.rot\_z = 0

**class** image:

**def** \_\_init\_\_(self, resolution, fieldOfView, polarPoint):

fovDegrees = fieldOfView \* 180 / math.pi

self.x = (resolution[0] / 2) + (resolution[0] \* polarPoint.rot\_y / fovDegrees)

*#As my image is approximated to a circle instead of a rectangle (for ease of calculations),*

*#the x resolution is used for both.*

self.y = (resolution[1] / 2) + (resolution[0] \* polarPoint.rot\_x / fovDegrees)

**class** world:

*"""Their Z axis is our cameraNormal.*

*Their Y axis is our Z axis inverted.*

*Their X axis is perpendicular to both."""*

**def** \_\_init\_\_(self, cameraPosition, cameraNormal, point):

cameraToPoint = point - cameraPosition

*#Converting from the simulation's coordinate system to the camera's coordinate system.*

cameraZAxis = cameraNormal

cameraYAxis = Vector3(0, 0, -1)

cameraXAxis = cameraYAxis.cross(cameraZAxis)

self.x = cameraXAxis.dot(cameraToPoint)

self.y = cameraYAxis.dot(cameraToPoint)

self.z = cameraZAxis.dot(cameraToPoint)

**class** polar:

**def** \_\_init\_\_(self, worldPoint):

worldVector = Vector3(worldPoint.x, worldPoint.y, worldPoint.z)

self.length = worldVector.magnitude

self.rot\_x = (math.atan2(worldPoint.y, worldPoint.z) / math.pi) \* 180

self.rot\_y = (math.atan2(worldPoint.x, worldPoint.z) / math.pi) \* 180

**class** Point:

**def** \_\_init\_\_(self, resolution, fieldOfView, cameraPosition, cameraNormal, point):

self.world = world(cameraPosition, cameraNormal, point)

self.polar = polar(self.world)

self.image = image(resolution, fieldOfView, self.polar)

**class** MarkerInfo:

**def** \_\_init\_\_(self, id, size, teamNumber):

self.code = id

self.size = size

**if** id <= 23:

self.marker\_type = MARKER\_ARENA

self.token\_type = TOKEN\_NONE

self.offset = id

**elif** id >= 32 **and** id <= 41:

self.marker\_type = MARKER\_TOKEN

self.token\_type = TOKEN\_ORE

self.offset = id - 32

**else**:

goldOffset = id - 42

self.marker\_type = MARKER\_TOKEN

goldTeam = id // 4

**if** goldTeam == teamNumber:

self.token\_type = TOKEN\_GOLD

**else**:

self.token\_type = TOKEN\_FOOLS\_GOLD

self.offset = goldOffset % 4

**class** Marker:

**def** \_\_init\_\_(self, resolution, fieldOfView, cameraPosition, cameraNormal, currentTimestamp, teamNumber, markerDictionary):

self.info = MarkerInfo(markerDictionary["Id"], markerDictionary["Size"], teamNumber)

markerCentrePoint = 0.5 \* ( constructFromDictionary(markerDictionary["Corners"][0]) + constructFromDictionary(markerDictionary["Corners"][2]) )

self.centre = Point(resolution, fieldOfView, cameraPosition, cameraNormal, markerCentrePoint)

self.vertices = []

**for** corner **in** markerDictionary["Corners"]:

self.vertices.append(Point(resolution, fieldOfView, cameraPosition, cameraNormal, constructFromDictionary(corner)))

self.orientation = Orientation(markerDictionary["Corners"], cameraNormal)

self.res = resolution

self.timestamp = currentTimestamp

@property

**def** dist(self):

**return** self.centre.polar.length

@property

**def** rot\_y(self):

**return** self.centre.polar.rot\_y

**class** Motors:

*"""An interface for the motors in the RobotService."""*

**def** \_\_init\_\_(self, robotService):

self.\_robotService = robotService

**def** \_\_getitem\_\_(self, index):

**return** self.\_robotService.getMotorPower(index)

**def** \_\_setitem\_\_(self, index, value):

**return** self.\_robotService.setMotorPower(index, value)

**class** Robot:

*"""An interface for this robot's RobotService.*

*Attempts to accurately replicate the API of a real robot's Robot class, to allow*

*programs written for real robots to work with this class."""*

**def** \_\_init\_\_(self):

*"""Identifies the url of the RobotService based on the arguments passed to it when the program was initialised.*

*Then creates an interface with that service to match the robocon API,*

*and yields the program until the RobotService returns."""*

parser = argparse.ArgumentParser("RobotClient")

parser.add\_argument("--url", action="store", help="The URL of of the RobotThread's xmlrpc server.")

arguments = parser.parse\_args()

\_trace("Connecting to RobotService:")

self.\_robotService = xmlrpc.client.ServerProxy(arguments.url)

self.motors = Motors(self.\_robotService)

self.gpio = []

self.servos = []

self.zone = self.\_robotService.getTeamNumber()

\_trace("Connected, waiting for start.")

self.\_robotService.waitForStart()

\_trace("Starting.")

**def** print(self, message):

*"""Sends a message to be printed to the Controller's Standard Output."""*

**return** self.\_robotService.print(str(message))

**def** sleep(self, time):

*"""Yields the program until the specified number of seconds have passed in the simulation."""*

\_trace("Entering sleep.")

self.\_robotService.sleep(time)

\_trace("Exiting sleep.")

**def** see(self, res=(640, 480)):

*"""Yields the program for a brief amount of simulated time depending on the resolution specified,*

*and returns a list of visible marker objects. Raises an exception if an illegal resolution is specified."""*

legalResolutions = [

(640, 480),

(1296, 736),

(1296, 976),

(1920, 1088),

(1920, 1440)

]

**if** **not** ( res **in** legalResolutions ):

**raise** RuntimeError("Invalid resolution. Resolution must be one of (640, 480), (1296, 736), (1296, 976), (1920, 1088), (1920, 1440)")

visionDictionary = self.\_robotService.see(res)

markerObjects = []

**for** marker **in** visionDictionary["List of Markers"]:

markerObjects.append(Marker(

visionDictionary["Resolution"],

visionDictionary["Field of View"],

constructFromDictionary(visionDictionary["Camera Position"]),

constructFromDictionary(visionDictionary["Camera Normal"]),

visionDictionary["Timestamp"],

self.\_robotService.getTeamNumber(),

marker

))

**return** markerObjects

**def** waitForStart(self):

*"""Yields the program until the simulation begins."""*

**return** self.\_robotService.waitForStart()

## SimDisplay.py

**import** pygame

**from** pygame.locals **import** \*

**import** SimBase

**class** Display:

*"""A class that provides a display of the simulation and encapsulates the dependencies on the pygame module."""*

\_teamColourDictionary = {

0 : Color("Gold"),

1 : Color("Lime Green"),

2 : Color("Red"),

3 : Color("Blue")

}

\_darkTeamColourDictionary = {

0 : Color(100, 80, 0),

1 : Color(0, 100, 0),

2 : Color(100, 0, 0),

3 : Color(0, 0, 100)

}

\_tokenTypeColourDictionary = {

"Ore" : Color("Grey"),

"Team 0 Gold" : Color("Gold"),

"Team 1 Gold" : Color("Lime Green"),

"Team 2 Gold" : Color("Red"),

"Team 3 Gold" : Color("Blue")

}

**def** \_\_init\_\_(self):

*"""Creates the window for the display, and populates it with the objects currently in the arena."""*

pygame.init()

pygame.display.set\_caption("Test program.")

self.screen = pygame.display.set\_mode( (620, 620), pygame.RESIZABLE )

*#The screen is set to slightly larger than 6m by 6m, to allow the arena walls to be displayed.*

self.clock = pygame.time.Clock()

self.updateDisplay()

**def** \_pymunkToPygame(self, point):

*"""Converts pymunk coordinates to pygame coordinates."""*

width, height = self.screen.get\_size()

**return** int((point.x + 3.1) \* width / 6.2), int((3.1 - point.y) \* height / 6.2)

**def** \_drawPoly(self, shape, colour, borderColour = **None**):

*"""Takes a pymunk Poly shape and draws the polygon in the specified colour,*

*with a border if the borderColour argument is set."""*

vertexes = shape.get\_vertices()

pygameVertexes = []

**for** vertex **in** vertexes:

worldVertex = shape.body.local\_to\_world(vertex)

pygameVertexes.append(self.\_pymunkToPygame(worldVertex))

pygame.draw.polygon(self.screen, colour , pygameVertexes)

**if** borderColour != None:

pygame.draw.polygon(self.screen, borderColour , pygameVertexes, 3)

**def** processInputs(self):

*"""Checks to see if the window has been closed or the Escape key has been pressed,*

*and if it has, sets the simulation time to the end of the simulation to end it prematurely.*

*Also rescales the display if the window size is changed."""*

**for** event **in** pygame.event.get():

**if** event.type == QUIT:

SimBase.theTime = SimBase.endTime

**elif** event.type == KEYDOWN **and** event.key == K\_ESCAPE:

SimBase.theTime = SimBase.endTime

**elif** event.type == VIDEORESIZE:

self.screen = pygame.display.set\_mode((event.w, event.h), pygame.RESIZABLE)

**def** updateDisplay(self):

*"""Updates the display to the current state of SimBase.space, and then waits*

*a variable amount of time to keep the framerate consistent at 64 fps"""*

self.screen.lock()

self.screen.fill((255,255,255))

*#Drawing is done in two passes to ensure the zones are drawn underneath the tokens or walls.*

**for** shape **in** SimBase.space.shapes:

**if** isinstance(shape.body, SimBase.Zone):

colour = Display.\_darkTeamColourDictionary[shape.body.teamNumber]

self.\_drawPoly(shape, colour)

**for** shape in SimBase.space.shapes:

**if** isinstance(shape.body, SimBase.Token):

mostRecentSeen = -5

bestBorderColor = **None**

**for** team **in** range(4):

**if** (SimBase.theTime - shape.body.lastSeenList[team] < 1) **and** (shape.body.lastSeenList[team] > mostRecentSeen):

bestBorderColor = Display.\_darkTeamColourDictionary[team]

mostRecentSeen = shape.body.lastSeenList[team]

self.\_drawPoly(shape, self.\_tokenTypeColourDictionary[shape.body.type], bestBorderColor)

**elif** isinstance(shape.body, SimBase.WallSegment):

bestColour = Color("Black")

mostRecentSeen = -5

**for** team **in** range(4):

**if** (SimBase.theTime - shape.body.lastSeenList[team] < 1) **and** (shape.body.lastSeenList[team] > mostRecentSeen):

bestColour = Display.\_teamColourDictionary[team]

mostRecentSeen = shape.body.lastSeenList[team]

self.\_drawPoly(shape, bestColour)

**elif** isinstance(shape.body, SimBase.Robot):

self.\_drawPoly(shape, Display.\_teamColourDictionary[shape.body.teamNumber])

self.screen.unlock()

pygame.display.flip()

self.clock.tick(64)

# Test Files

## Vector3\_test.py

1. **import** unittest

**import** math

**from** vector3 **import** \*

**class** Vector3Test(unittest.TestCase):

**def** testStringRepresentation(self):

*"""Tests that a vector can be represented as a string, and that a vector is not equal to its string representation."""*

a = Vector3(1, 2, 3)

b = "Vector3(1, 2, 3)"

self.assertEqual(str(a), b)

self.assertNotEqual(a, b)

self.assertNotEqual(b, a)

**def** testNonIntegerStringRepresentation(self):

*"""Tests that a vector with arbitrary non integer components can be represented as a string."""*

a = Vector3(1.0, -2.5, math.pi)

b = "Vector3(1.0, -2.5, " + str(math.pi) + ")"

self.assertEqual(str(a), b)

self.assertNotEqual(a, b)

**def** testEquivalent(self):

*"""Tests that two Vector3s can compared."""*

a = Vector3(1, 2, 3)

b = Vector3(1, 2, 3)

c = Vector3(3, 2, 1)

self.assertEqual(a, b)

self.assertNotEqual(a, c)

**def** testAddition(self):

*"""Tests that two Vector3s can be added."""*

a = Vector3(1, 2, 3)

b = Vector3(4, 5, 6)

c = Vector3(5, 7, 9)

d = Vector3(0, 0, 0)

self.assertEqual(a + b, c)

self.assertNotEqual(a + b, d)

**def** testNegativeAddition(self):

*"""Tests that negative Vector3s can be added."""*

a = Vector3(-1, -2, -3)

b = Vector3(1, 2, 3)

c = Vector3(0, 0, 0)

d = Vector3(2, 4, 6)

self.assertEqual(a + b, c)

self.assertNotEqual(a + b, d)

**def** testUnaryNegative(self):

*"""Tests that the unary negative function multiplies the vector by a scalar -1."""*

a = Vector3(-1, -2, -3)

b = Vector3(1, 2, 3)

self.assertEqual(a, -b)

self.assertNotEqual(a, -a)

**def** testSubtraction(self):

*"""Tests that two Vector3s can be subtracted."""*

a = Vector3(5, 7, 9)

b = Vector3(1, 2, 3)

c = Vector3(4, 5, 6)

d = Vector3(0, 0, 0)

self.assertEqual(a - b, c)

self.assertNotEqual(a - b, d)

**def** testNegativeSubtraction(self):

*"""Tests that negative Vector3s can be subtracted."""*

a = Vector3(1, 2, 3)

b = Vector3(-1, -2, -3)

c = Vector3(2, 4, 6)

d = Vector3(0, 0, 0)

self.assertEqual(a - b, c)

self.assertNotEqual(a - b, d)

**def** testScalarMultiply(self):

*"""Tests that vectors can be multiplied by a scalar."""*

a = Vector3(1, 2, 3)

b = Vector3(3, 6, 9)

c = Vector3(1, 2, 3)

self.assertEqual(3 \* a, b)

self.assertEqual(a \* 3, b)

self.assertNotEqual(3 \* a, c)

**def** testZeroScalarMultiply(self):

*"""Tests that vectors can be multiplied by the scalar 0."""*

a = Vector3(1, 2, 3)

b = Vector3(0, 0, 0)

self.assertEqual(0 \* a, b)

self.assertNotEqual(0 \* a, a)

**def** testDecimalScalarMultiply(self):

*"""Tests that vectors can be multiplied by a non integer scalar."""*

a = Vector3(1, 2, 3)

b = Vector3(1.5, 3, 4.5)

c = Vector3(2, 3, 5)

self.assertEqual(1.5 \* a, b)

self.assertEqual(a \* 1.5, b)

self.assertNotEqual(1.5 \* a, c)

**def** testMagnitude(self):

*"""Tests that the magnitude of a vector can be correctly calculated."""*

a = Vector3(1, 2, 3)

b = Vector3(-1, -2, -3)

c = Vector3(0, 0, 0)

d = Vector3(3, 4, 12)

self.assertAlmostEqual(a.magnitude, math.sqrt(14))

self.assertEqual(a.magnitude, b.magnitude)

self.assertEqual(c.magnitude, 0)

self.assertEqual(d.magnitude, 13)

**def** testUnitVector(self):

*"""Tests that the unit vector corresponding to a vector can be correctly calculated."""*

a = Vector3(3, 4, 12)

b = Vector3(3/13, 4/13, 12/13)

c = Vector3(0, 0, 0)

self.assertAlmostEqual(a.unit, b)

self.assertEqual(a.unit.magnitude, 1)

**with** self.assertRaises(ZeroDivisionError):

c.unit

**def** testDot(self):

*"""Tests that the dot product of a vector can be correctly calculated."""*

a = Vector3(1, 2, 3)

b = Vector3(2, 4, 6)

c = Vector3(1, 2, 0)

d = Vector3(2, -1, 0)

e = Vector3(1, 0, 0)

f = Vector3(1, 1, 0)

*#This test caught a typing error.*

self.assertAlmostEqual(a.dot(b), a.magnitude \* b.magnitude) *#For parallel vectors, a.b = |a|\*|b|*

self.assertEqual(c.dot(d), 0) *#For perpendicular vectors, c.d = 0*

self.assertAlmostEqual(e.dot(f), e.magnitude \* f.magnitude \* math.cos(0.25\*math.pi) ) *#e.f = |e|\*|f|\*cos(angle)*

**def** testCross(self):

*"""Tests that the cross product of a vector can be correctly calculated."""*

x = Vector3(1, 0, 0)

y = Vector3(0, 1, 0)

z = Vector3(0, 0, 1)

zero = Vector3(0, 0, 0)

a = Vector3(1, 1, -2)

b = Vector3(-5, 1, 3)

c = Vector3(5, 7, 6)

self.assertEqual(x.cross(y), z)

*#This test caught a typing error.*

self.assertEqual(y.cross(x), -z)

self.assertEqual(x.cross(x), zero)

self.assertEqual(x.cross(zero), zero)

self.assertEqual(a.cross(b), c)

**def** testAngleBetween(self):

*"""Tests that the angle between two vectors can be correctly calculated."""*

*#Vector facing up.*

VectorA = Vector3(0, 0, 1)

*#Vectors perpendicular:*

VectorB = Vector3(1, 0, 0)

VectorC = Vector3(-1, 0, 0)

VectorD = Vector3(0, 1, 0)

VectorE = Vector3(0, -1, 0)

self.assertAlmostEqual( VectorA.angleBetween(VectorB), math.pi / 2 )

self.assertAlmostEqual( VectorA.angleBetween(VectorC), math.pi / 2 )

self.assertAlmostEqual( VectorA.angleBetween(VectorD), math.pi / 2 )

self.assertAlmostEqual( VectorA.angleBetween(VectorE), math.pi / 2 )

*#Vector facing down.*

VectorF = Vector3(0, 0, -1)

self.assertAlmostEqual( VectorA.angleBetween(VectorF), math.pi )

self.assertAlmostEqual( VectorF.angleBetween(VectorA), math.pi )

*#Longer vector at 45 degrees.*

VectorG = Vector3(0, 1, 1)

*#This test caught a typing error.*

self.assertAlmostEqual( VectorA.angleBetween(VectorG), math.pi / 4)

self.assertAlmostEqual( VectorG.angleBetween(VectorA), math.pi / 4)

**def** testRotateAroundZ(self):

*"""Tests that a vector can be correctly rotated by an angle."""*

VectorA = Vector3(0.125, 0, 0)

VectorB = Vector3(0, 0.125, 0)

VectorC = Vector3(-0.125, 0, 0)

VectorD = Vector3(0, -0.125, 0)

*#it is impossible to assert a vector is almost equal to a vector, so the vectors are subtracted*

*#from one another and the magnitude is taken*

self.assertEqual( (VectorA.rotateAroundZ(0) - VectorA).magnitude, 0 )

self.assertAlmostEqual( (VectorA.rotateAroundZ(math.pi / 2) - VectorB).magnitude, 0 )

self.assertAlmostEqual( (VectorA.rotateAroundZ(-math.pi) - VectorC).magnitude, 0 )

self.assertAlmostEqual( (VectorA.rotateAroundZ(-math.pi / 2) - VectorD). magnitude, 0 )

**class** PlaneTest(unittest.TestCase):

**def** testPlaneFacingCamera(self):

*"""Tests that it can be calculated if a plane is facing the camera."""*

planeA = Plane( Vector3(0, 0, 0), Vector3(1, 0, 0), Vector3(0, 1, 0) ) *#facing upwards*

cameraPositionA = Vector3(0, 0, 1)

cameraPositionB = Vector3(0, 0, 0) *#inside the planes (does not face either)*

planeB = Plane( Vector3(0, 0, 0), Vector3(1, 0, 0), Vector3(0, -1, 0) ) *#facing downwards*

self.assertTrue( planeA.isFacingCamera(cameraPositionA) )

self.assertFalse( planeA.isFacingCamera(-cameraPositionA) )

self.assertFalse( planeA.isFacingCamera(cameraPositionB) )

self.assertFalse( planeB.isFacingCamera(cameraPositionB) )

**def** testPlaneObstructingPoint(self):

*"""Tests that it can be calculated if a bounded plane obstructs the line between two points."""*

*#A square normal to the Z axis, centred on 0, 0, 0, with side length 2.*

plane = Plane( Vector3(-1, -1, 0), Vector3(1, -1, 0), Vector3(-1, 1, 0) )

*#A point above the plane.*

cameraPositionA = Vector3(0, 0, 1)

*#A point directly under the camera, below the plane.*

pointA = Vector3(0, 0, -1)

*#A point directly under the camera, above the plane.*

pointB = Vector3(0, 0, 0.5)

*#A point directly above the camera, above the plane.*

pointC = Vector3(0, 0, 2)

*#A point far beyond the bounds of the plane, below the plane.*

pointD = Vector3(0, 10, -1)

self.assertTrue( plane.isObstructingPoint(pointA, cameraPositionA) )

self.assertFalse( plane.isObstructingPoint(pointB, cameraPositionA) )

self.assertFalse( plane.isObstructingPoint(pointC, cameraPositionA) )

self.assertFalse( plane.isObstructingPoint(pointD, cameraPositionA) )

*#A point to the side of the plane.*

cameraPositionB = Vector3(0, 2, 0)

*#A point on the other side of the plane.*

pointE = Vector3(0, -2, 0)

self.assertFalse( plane.isObstructingPoint(pointE, cameraPositionB) )

## TestProgramLazy.py

1. **import** RobotClient

**import** random

print("LazyRobot started!")

robot = RobotClient.Robot()

**while** **True**:

robot.sleep( random.randint(1, 20) )

## TestProgramDrivey.py

1. **import** RobotClient

**import** random

print("DriveyRobot started!")

robot = RobotClient.Robot()

robot.motors[1] = 100

robot.motors[2] = 100

robot.sleep( 180 )

## TestProgramCrashy.py

1. **import** RobotClient

**import** random

print("DriveyRobot started!")

robot = RobotClient.Robot()

print(1 / 0)

## TestProgramSee.py

**import** RobotClient

**import** random

print("LazyRobot started!")

robot = RobotClient.Robot()

**while** **True**:

robot.sleep(5)

seen = robot.see( (1920,1440) )

**for** marker **in** seen:

print("Marker " + str(marker.info.code) + ":")

**for** i **in** range(0,4):

print(" Vertex[{}]: image=({},{})".format(i, marker.vertices[i].image.x, marker.vertices[i].image.y))

print(" Vertex[{}]: polar=({},{})".format(i, marker.vertices[i].polar.rot\_x, marker.vertices[i].polar.rot\_y))

print(" Vertex[{}]: world=({},{},{})".format(i, marker.vertices[i].world.x, marker.vertices[i].world.y, marker.vertices[i].world.z))

print(" dist={}".format(marker.dist))

print(" rot\_y={}".format(marker.rot\_y))

print(" Orientation=({},{},{})".format(marker.orientation.rot\_x, marker.orientation.rot\_y, marker.orientation.rot\_z))

robot.sleep(200)

## TestProgramClient.py

**import** RobotClient **as** robot

MULTIPLIER\_LEFT = 2

MULTIPLIER\_RIGHT = 2

STARTUP\_TIME = 0

SPEED\_50 = 0.4

SPEED\_ANGULAR\_30 = 120

R = robot.Robot()

**def** move(distance):

R.motors[1] = MULTIPLIER\_LEFT \* 50

R.motors[2] = MULTIPLIER\_RIGHT \* 50

R.sleep(STARTUP\_TIME)

R.sleep(distance / SPEED\_50)

R.motors[1] = 0

R.motors[2] = 0

**def** turn(degrees):

multiplier = 1

**if** degrees < 0:

multiplier = -1

R.motors[1] = MULTIPLIER\_LEFT \* 30 \* multiplier

R.motors[2] = MULTIPLIER\_RIGHT \* -30 \* multiplier

R.sleep(abs(degrees) / SPEED\_ANGULAR\_30)

R.motors[1] = 0

R.motors[2] = 0

R.sleep(1)

**while** **True**:

R.print("Looking!")

markers = R.see()

**if** len(markers) == 0:

turn(45)

**else**:

**for** marker **in** markers:

**if** marker.info.marker\_type == robot.MARKER\_TOKEN:

R.print(marker.info.code)

turn(marker.rot\_y)

R.print(marker.rot\_y)

move(marker.dist-0.1)

**break**

**else**:

turn(45)