Self-Driving Cars for Autonomous Intersection Management

CARS

Jordan Jooste JSTJOR001, Jonathon Weideman WDMJON001, Robbie Perrot PRRROB006

Contents

Abstract

This report is to document the re-implementation of the Austin Universities Autonomous Intersection Manager (AIM) simulator. AIM controls self driving vehicles in an intersection to optimally get each vehicle through without crashes. Cars send a list of proposed routes and the intersection manager (IM) works out the most optimal route for the vehicle to travel through by accepting or rejecting them. This simulator has been tested in real world self driving vehicles to control intersection management.

However, it has been developed by multiple people and had additional classes added on as a result, which makes it complicated to understand. The original simulator managed intersections with autonomous vehicles(self driving cars) using a First-Come-First-Served policy.

The re-implementation (AIM5) is more efficient in some ways than the original and added the option of including pedestrians for vehicles to avoid. These pedestrians spawn randomly and cross the intersection when allowed by a traffic light system lights controlled by a pedestrian policy and triggered by the pedestrians pushing a button. The new AIM is designed to be more easily extendable and will perform well on multi-core computers and one of the main optimizing factors is achieved by threading steps in the system.

Glossary

ACZ - Admission Control Zone
AIM - Automatic Intersection Manager
FCFS- First Come, First Served
GUI - Graphic User Interface
IM - Intersection Manager VIN - Vehicle Identification Number

Introduction

Technology advances in self driving vehicles have been paving the way for future vehicle engineering and road engineering to accommodate for these changes. We will implement an Autonomous Intersection Management controller that safely and effectively directs traffic through a set of intersections in a simulated network of roads. This AIM system will guide autonomous vehicles through an intersection in the most optimal fashion while adhering to a First Come First Served (FCFS) policy. This aims to replace traditional traffic lights and upgrade the system to one which will increase throughput with coded algorithms.

The implementation is built on an existing simulator built by researchers in the computer science department from the University of Austin Texas(AIM4). The code consisted of a multitude of classes which have been used and updated and added on to the original code. Many hours of the development of the new program went to reading and understanding the source code. This code was optimized for efficiency by threading multiple steps and methods within the classes.

Secondly, we re-implemented the AIM controller to include a dynamic control module for the autonomous vehicles, allowing the vehicles to safely and effectively navigate the roads and intersections under conditions of imperfect traffic flow and incomplete information. Specifically, the AIM controls the flow of vehicles to avoid any pedestrian collisions. The simulator GUI has a slider panel at start up to set preferences for the simulator variables when run. The final product was created by an iterative process of coding and testing to be sure it worked with the original code. We worked off a horizontal prototype and added functionality to existing classes and created new ones as we needed.

Requirements Captured

Functional Requirements

- GUI shows a map with roads.
- Vehicles spawned and act independently.
- Vehicles sent through intersection when safe.
- Vehicles do not crash into other vehicles or pedestrians.
- Vehicles stop to allow pedestrians to cross intersection.
- Pedestrians push button when spawned.
- Pedestrians cross intersection when safe.
- Vehicles continue to cross when pedestrians have finished crossing.
- Vehicles travel through intersection based on a first come first served policy.
- Pedestrians must cross after max wait time exceeded.
- Vehicles can continue to turn if there are pedestrians using another lane.
- Pedestrians do not push button if button already pushed.
- Pedestrians all move across intersection at same rate.

Non-functional requirements:

- Performance enhanced by threading.
- Vehicles stopping for pedestrians optimized by calculations.
- Maintainability created by using well commented code.
- Extendability to other problems.

Usability requirements:

- User interface is easy to use.
- User interface is interesting but neat.
- Buttons clear and easy to find.

- Can be run on multiple systems.
- Information on each vehicle easy to access.
- Sliders can be adjusted for customization of simulation.
- Changes to system can be made while running the program.
- Statistics of the simulation is visible during and after the simulation.

High Level Use Case Diagrams

The following use case diagrams show the typical flow of events and the alternative flow of events in the simulator system.

Vehicle to Intersection Manager Use Cases

Use Case 1

Goal in context

Vehicle spawned and drives through intersection without crashes.

Primary Actor

Vehicle

Secondary Actor

Intersection Manager

Precondition

Simulation is running

Typical Flow of Events

Vehicle Action	Intersection Response
1. Vehicle spawned	-
2. Check if vehicle in front	3. No vehicle in front
4. If at max speed, maintain max speed	-
5. Vehicle sends requests to intersection manager	6. Intersection manager checks all path requests 7. Intersection Manager accepts a path request
8. Vehicle travels through intersection	-

Alternative Flow of Events 1

Vehicle Action	Intersection Response
-	3.1 There is a vehicle in front
4.1 Decrease speed to avoid collision	-
Continue to primary flow step 5	-

Alternative Flow of Events 2

Vehicle Action	Intersection Response
4. If not at maximum speed, accelerate until at maximum	-

Vehicle Action	Intersection Response
5. Back to primary flow step 5	-

Alternative Flow of Events 3

Vehicle Action	Intersection Response
-	6.1 Pedestrians crossing over road on desired route (top,bottom,left,right)
-	7.1 Intersection manager rejects path request
8. Vehicle waits	-
Continue to primary flow step 5	-

Alternative Flow of Events 4

Vehicle Action	Intersection Response
-	6.1 Pedestrians crossing over road on desired route (Diagonal bottom left to right)
-	7.1 Vehicles going straight rejected
-	7.2 Vehicle in rightmost lane turning to the right accepted
Continue to primary flow step 5	-

Alternative Flow of Events 5

Vehicle Action	Intersection Response
-	6.1 Pedestrians crossing over road on desired route (Diagonal bottom right to left)
-	7.1 Vehicles going straight rejected
7.2 Vehicle in leftmost lane turning to the left accepted	
Continue to primary flow step 5	

Alternative Flow of Events 6

Vehicle Action	Intersection Response
-	7.1 IM sends message asking for vehicle to re request path as pedestrians ahead
8.1 Vehicle checks if it can slow down	9.1 Enough time to stop before intersection
10.1 Vehicle slows down and stops before intersection	-

Vehicle Action	Intersection Response
Continue to primary flow step 5	

Alternative Flow of Events 7

Vehicle Action	Intersection Response
-	7.1 IM sends message asking for vehicle to rerequest path as pedestrians ahead
8.1 Vehicle checks if it can slow down	9.1 Not enough time to stop before intersection
10.1 Vehicle proceeds on route through intersection	-

Alternative Flow of Events 8

Vehicle Action	Intersection Response
-	6.1 Vehicle with request accepted to be crossing over desired route and collision detected.
-	7. Intersection manager rejects path request
Continue to primary flow step 5	

Pedestrian to Intersection Manager Use Cases

Use Case 1

Goal in context

Pedestrians to spawn and cross the intersection when allowed by the pedestrian light.

Primary Actor

Pedestrian

Secondary Actor

Intersection Manager

Preconditions

Simulation is running

Pedestrian slider set above 0

Typical Flow of Events

Pedestrian Action	Intersection Response
1. Pedestrian spawned	-
2. Pushes button if button is off	3. Displays wait signal
-	4. Start timer
-	5. Max time reached
6. Display walk signal	
8. Pedestrian crosses intersection	-

Alternative Flow of Events 1

Pedestrian Action	Intersection Response
2.1 Pedestrian waits if button is already on	-
Continue to primary flow step 5	-

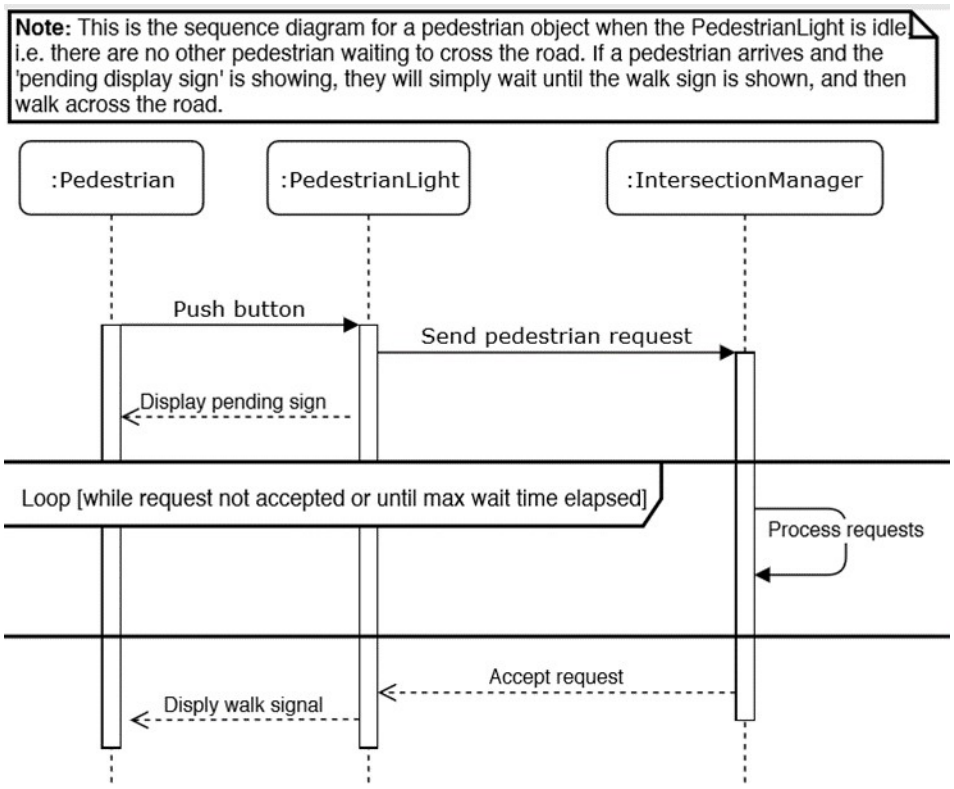


Figure 1: Sequence Diagram for Pedestrain Lights

Design Overview

The design of the new implementation of the AIM4 code was originally planned to be a complete rewrite of the code from the beginning while using key aspects of the AIM4 code. However after analyzing the code it was determined that it would not be feasible with our time schedule and decided to use the base code and add functionality and optimize it. This gave us a horizontal-vertical prototype hybrid with most features already implemented and a few that needed much enhancing and improving. Our prototype was evolutionary as it maintained much of the same code through to the final product.

Our team used some of the scrum practices while working on the project. We had stand up group meetings and used a Gantt chart to monitor our progress and ensure we stayed on schedule. We met most of our self imposed deadlines and this kept us on track to completing within the given time. One risk of this complex task was the short amount of time given. We worked around this by implementing our functions on top of the existing code.

Architecture Used

The original AIM project contained multiple classes which had to be investigated and understood before changing the code to suit our scope. We did not update and change every class but changes made will be detailed below. In order to understand the complete system an overview is needed. The system's architecture is a hybrid of the Model View Controller architecture. The View can be seen as the GUI and the controller sets up the simulation and roads etc. from the inputs selected on the GUI. The Model is the rest of the programs elements that make up the Simulator. The controller updates the GUI and the model.

The input to the view causes the controller to notify the model and changes are represented on the GUI. This architecture was needed as there are many classes that interact and the GUI view can be altered during

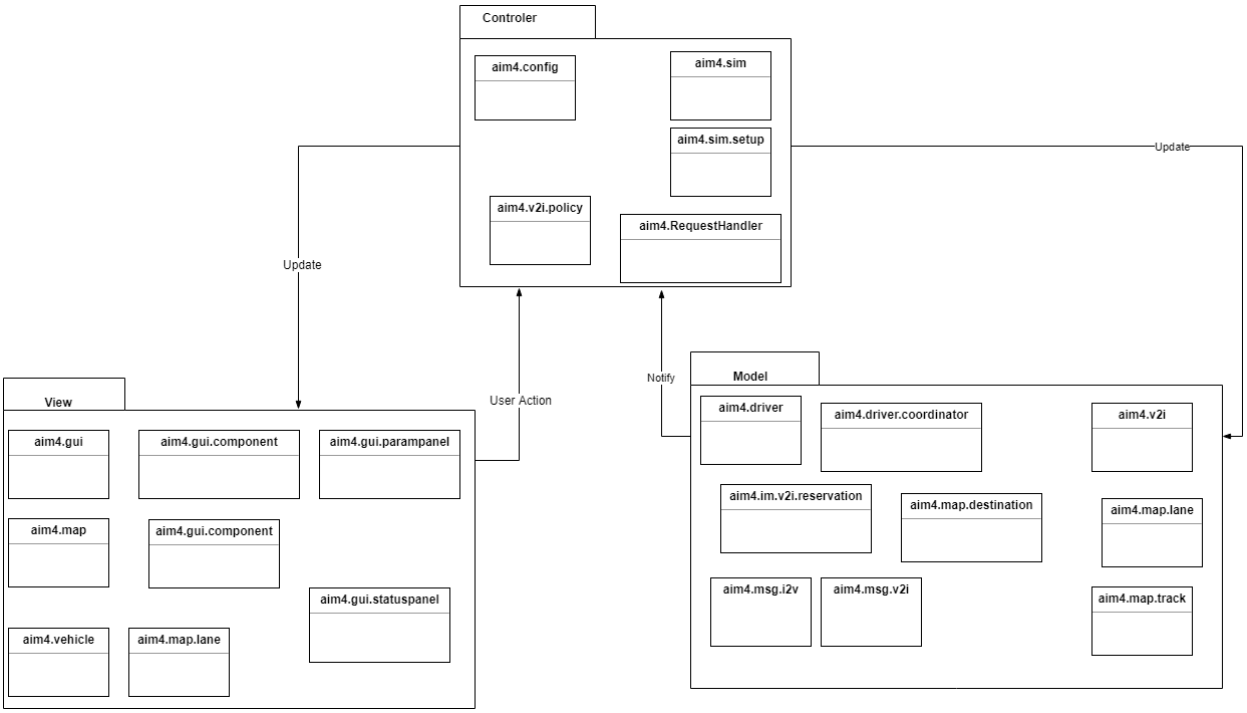


Figure 2: Architecture Diagram

the simulation and must reflect immediately. The "separation" of the parts makes it easier to understand the system as a whole and to change it if necessary. It can be called a hybrid architecture as many classes share relationships and elements across the separate parts so are not completely independent. See Figure 1 for a high level view of the architecture used. There are many relationships between the model, view and controller and so complete view of every package and its dependencies can be found in appendix A.

Data Organization

Directory, Folder, and File Naming

For our data organization of our classes we were limited in our choice of naming convention as we had to adhere to the original format. Our classes are all named with aim4.(more descriptive name of class).java. We chose to code in Java and used Github as a version control and used it to manage our development and be able to code individually but as a team.

Algorithms Used

One of the featured algorithmic patterns used in this project is the use of the FCFS algorithm, used for the intersection manager. This algorithm strictly serves vehicles in the order in which they arrive or request a path. If the path requested is rejected, they must reapply and are dealt with after vehicles which have applied before. When a vehicle sends a request it contains multiple proposals for routes it wishes to take and the intersection manager accepts one or rejects them all.

Hashmaps were also a featured algorithm choice, this is because of their ability to search and find items quickly. Most of the AIM4's job is to search quickly and make decisions so time efficiency is important. Hashmaps work quickly and efficiently as they use key and value pairs to store information together to aid in recovery of information. An example is the vinToReservationId Hashmap that maps a vehicle's vin to its reservation.

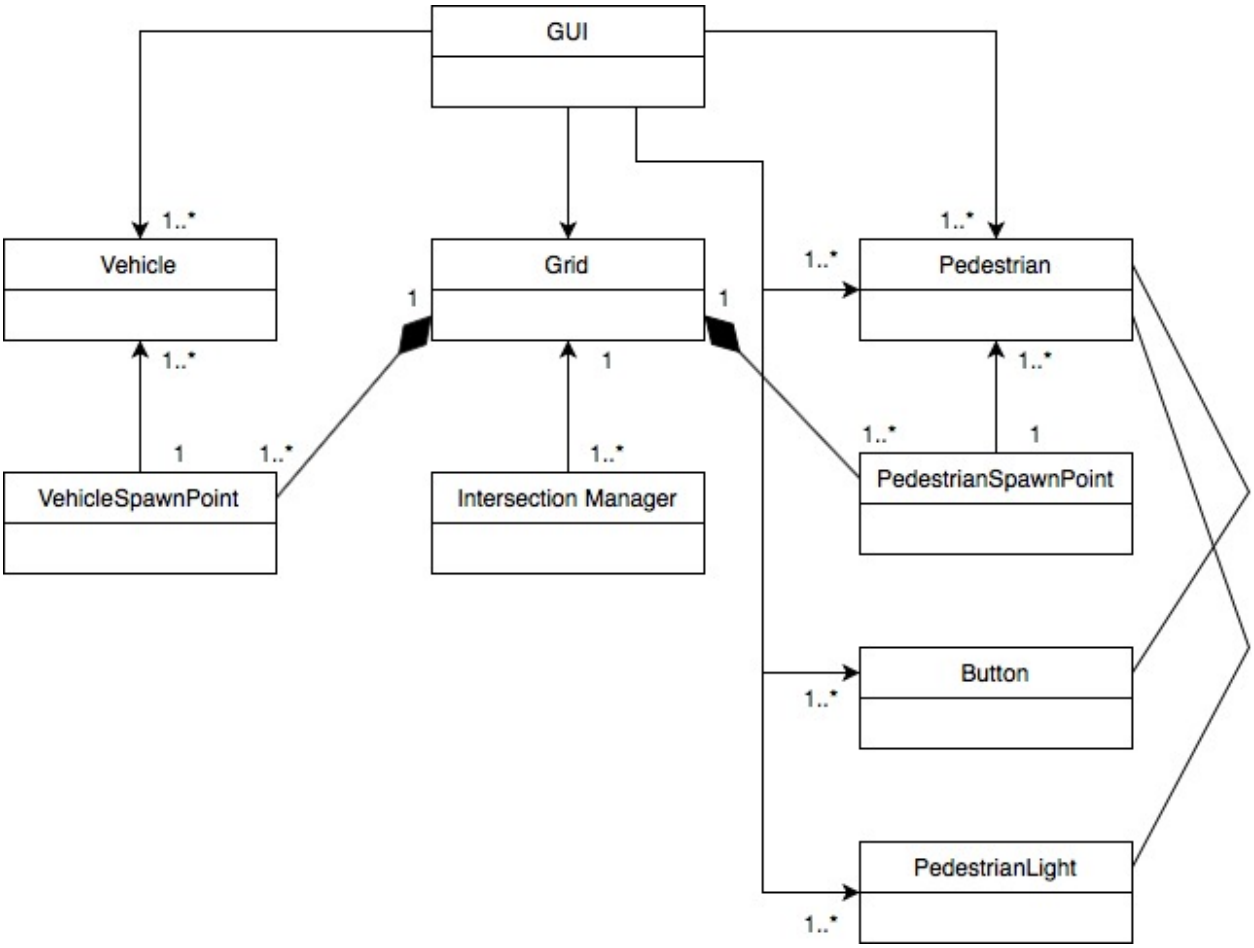


Figure 3: Design Class Diagram

To optimize the code we used threading within existing classes to increase efficiency. Threading allows the vehicles to spawn and act in parallel which will optimize the code for large scale use and for future implementations.

Implementation

What we added to AIM4

Our aim was to implement pedestrian functionality and optimize FCFS within the AIM program given. We created our own classes, changed some existing classes and added our own methods to do so.

Overview of the changes to vehicles: They needed to "view pedestrians" and stop when they were crossing. We implemented a system by which the vehicle (even if path already accepted from IM) would slow down and stop if the pedestrians would be crossing over the path needed. They then, reapply for a path through the intersection. The vehicles follow a specific policy we created to control the movement of vehicle and pedestrians.

Overview of the changes to Pedestrians: We created a pedestrian policy to use a pedestrian request handler to stop vehicles from being able to cross the path that pedestrians are following. The lines the pedestrians follow show up on the GUI when the pedestrian policy is turned on for an intersection. The spawning of the pedestrians can also be seen by the white blocks turning pink as the pedestrians spawn.

Interaction Between Pedestrain Light and Vehicles The pedestrian light does not know if vehicles are around. When a button is pressed by a pedestrian, the light informs the intersection manger that it wants to turn on. The IM keeps track of all vehicles coming towards the intersection but have not gone through yet. Once through the intersection they send a message the IM. When a light needs to turn on, vehicles need to be told to recalculate paths, even if requests already exist and have been accepted. All of the vehicle IDs that are approaching the intersection are added to an outbox in the IM. The simulator then sends messages from the IM to each vehicle to recalculate their route based on the new traffic flow requirements. If their path is not available, they have to see if they can slow down in time before the intersection. This is done by calculating the requirements to slow down based on the stopping distance from the intersection line, their current velocity and the max deceleration to see if it can stop. If it cannot stop it proceeds through the intersection.

getLaneCardinalDirection: Added the functionality to determine which direction the vehicle is turning from the given input lane and output lane and assign it an understandable value such as North, South, East and West. The request handler has a copy of the intersection and receives requests which contain an arrival lane ID and a destination lane ID. We added methods to determine where the vehicle is coming from based on lane IDs.

calcTurnDirection method added to calculate the turn direction based off the lane IDs. Returns, left, right, straight or U-Turn.

StatusPanelContainer: A tabbed panel at the bottom of the GUI. The panel shows statistics and status information of the simulation, and contains buttons that can be used to change the behavior of the simulation.

PedestrianButtonPanel: A tab we added to the StatusPanelContainer object. In the tab there are 7 JButtons: "Left", "Right", "Top", "Bottom", "TopLeftToBottomRight", "TopRightToBottomLeft", and "Stop All". These buttons will be "pushed" by virtual pedestrians who wish to cross the road. For example, a pedestrian wishing to cross over the left hand side of the intersection will push the "Left" button. The Intersection Manager will then prevent any traffic from passing through the left lanes, allowing the pedestrian(s) to safely pass through.

PedestrianRequestHandler: A class that specifies the policy an intersection must follow when a given configuration of pedestrians are present. If the slider is above 0 for pedestrians this policy will turn on. It contains a series of booleans (for example "left", "bottom", or "topLeftToBottomRight") that are set to true when pedestrians are crossing that part of the road. If, for example, the booleans "left" and "right" are true, the pedestrians will be able to cross the left and right hand side verticals of the intersection, while vehicles will only be allowed to drive from the top lanes to the bottom. We plan on changing this class later on in the project to handle more complex kinds of intersections than the current 4 way intersection.

AutoDriverOnlySimulator: The version of the simulator that only allows for autonomous vehicles. We have edited this class by threading some of the key methods.

Viewer: The GUI of the simulator.

Class member functions

PedestrianButtonPanel:

PedestrianButtonPanel(Viewer viewer):

- A constructor class that returns a PedestrianButtonPanel object.
- Creates a tab called "Pedestrian Buttons" on the status panel in the GUI.
- Adds the 7 JButtons to the tab
- Adds ActionListener objects to each button in order to initiate a reaction when the user clicks one of the buttons.

Pedestrian:

- Currently an empty class
- Could be extended to be more realistic on the GUI (move around etc)

actionPerformed(ActionEvent e)

- Responds to the clicking of one of the pedestrian buttons.
- Identifies the source of the action event (e.g. the "Top" button), and then changes the value of the corresponding boolean (e.g. "top") in the PedestrainRequestHandler by calling one of the setter methods (e.g. "setLeft()").

PedestrianRequestHandler:

Setters (e.g. setLeft()):

- Changes the boolean value of a pedestrian path. If the boolean is false, it is set to true. If it is true it is set to false. In terms of the simulation this would be equivalent to opening and closing the lanes along a pedestrians path. act()
- Lets the request handler act for a given time interval. processRequestMsg()
- Handles requests from cars wishing to pass through the intersection
- Rejects requests from vehicles that already have a reservation
- Filters through the proposals from vehicles and rejects those that are requesting an incorrect turning direction or those for which the necessary space-time is not available. removeProposalWIthIncorrect-TurnDirection()
- Remove proposals whose arrival (the exit from the intersection) time is prohibited by the RequestHandler objects policy.
- One pedestrian request handler per intersection, with 6 CrossWalk objects each (left, top, right, bottom, and 2 diagonals)
- If pedestrians are crossing a crosswalk, this class will deny requests by vehicles wishing to drive through this cross walk.

PedestrianSpawnPoint:

- Responsible for spawning pedestrians.
- Represents pedestrians on GUI by blocks (White if no pedestrians, pink when spawned)
- Communicates with crosswalk class to increment pedestrian counter

Act():

- For a certain probability of the time, say p, this method will create a new Pedestrian object and add it to the list of Pedestrians at the spawn point.
- For the other 1-p fraction of the time, nothing happens.
- This was done using a random number generator.

CrossWalk:

- Represents one of the 6 crosswalks in an intersection (four around the edges and 2 diagonals)
- Each cross walk has 3 possible states:
- Empty: No pedestrians are crossing or are about to cross.

- Waiting: Pedestrians are about to cross, but may have to wait a short amount of time before crossing to allow cars that cannot slow down in time to pass through the crosswalk. This corresponds to a yellow line in the GUI.
- Crossing: there are currently pedestrians crossing the cross walk. This corresponds to a red line in the GUI.
- This is where the timing of the pedestrians crossing is handled.
- A spawn point starts off empty
- When the first pedestrian spawns at a spawn point on either side of the cross walk, a timer set to maxWaitTime begins to elapse.

- This is the maxWaitTime that was fixed by one of the sliders in the input GUI.
- As this timer elapses, more pedestrians may spawn at either end of the cross walk.
- Once the timer reaches zero, the boolean value related to this crosswalk is set to true. Now, if a vehicle requests to pass through this cross walk, its request will be rejected.
- There is then a short window period that we called "waiting" where more pedestrians may still spawn while cars that cannot slow down in time pass through the intersection.
- Once this "waiting" period has elapsed, all pedestrians cross the cross walk for a certain amount of time.
- Once all pedestrians have crossed, the boolean value associated with this cross walk is set to false, and cars may pass through again.

Canvas:

drawCrossWalks():

- This method is called continuously by the updateCanvas() method which is responsible for updating the GUI.
- Our cross walks appear in 3 states:
- Empty or blank (no line): There are no pedestrians crossing the cross walk.
- Yellow: Pedestrians are about to cross, but they are waiting a short time to allow cars that cannot slow down fast enough to stop before the cross walk to pass through.
- Red: Pedestrians are crossing the cross walk.

drawPedestrianSpawnPoints()

- This method is also called continuously by updateCanvas().
- It colours each spawn point white if there are no pedestrians waiting at that spawn point, or pink if there are pedestrians waiting at the spawn point.

AutoDriversOnlySimulator:

letDriversAct():

- Uses the sensory information gathered in the previous step of the algorithm and the previous response from the Intersection Manager to plan future movement and to generate a request that will later be sent to the Intersection Manager.

moveVehicles():

- Moves the vehicle in the appropriate direction at the appropriate velocity and acceleration.

spawnPedestrians()

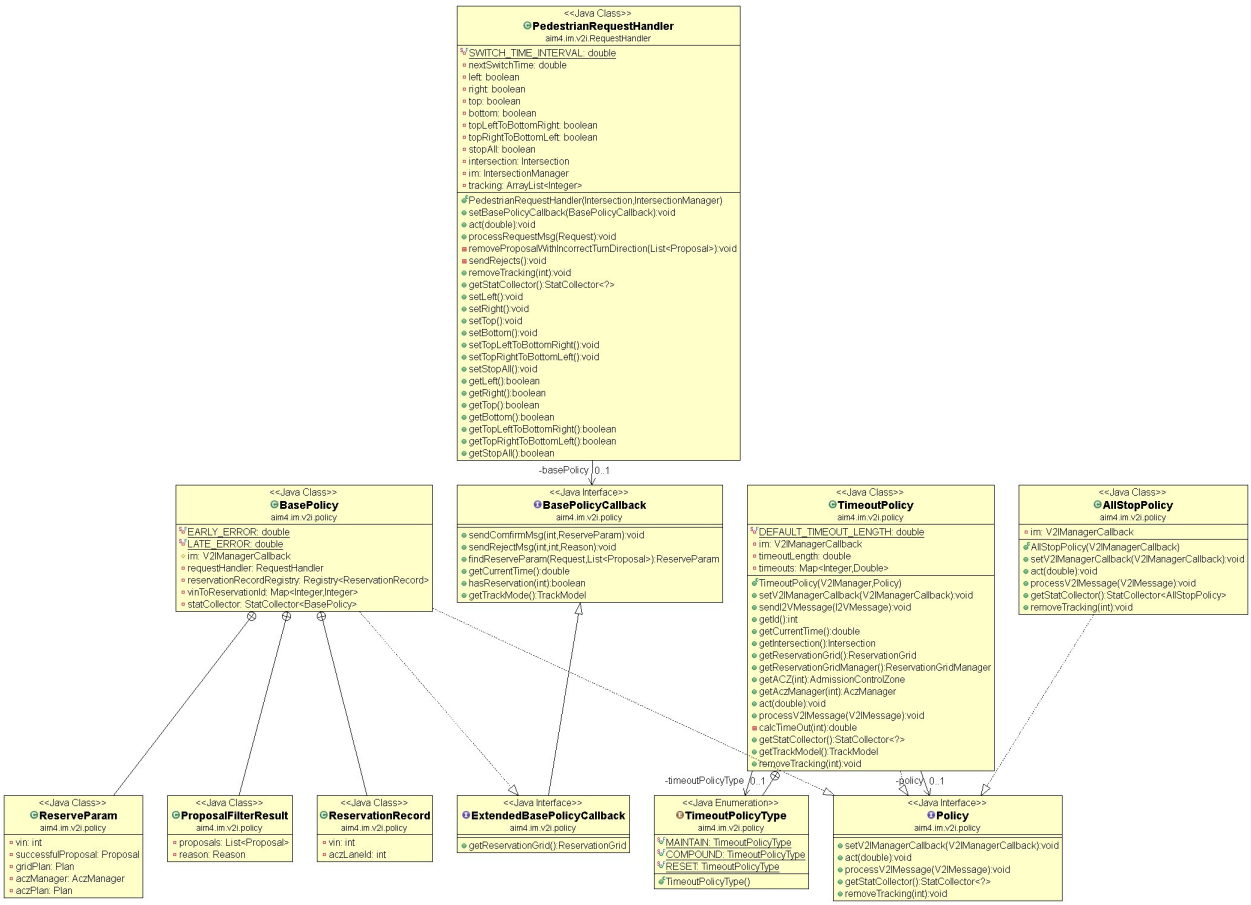


Figure 4: Request Relationship Diagram

- Each time this is called in the main algorithm, it rotates through all 12 spawn points in the intersection and calls act() on each spawn point.

CrossWalksAct

StatPanel:

- Displays statistics
- Edited to show total number of pedestrians that have crossed through all intersections, as well as the average wait time that each pedestrian waits at a spawn point before crossing.

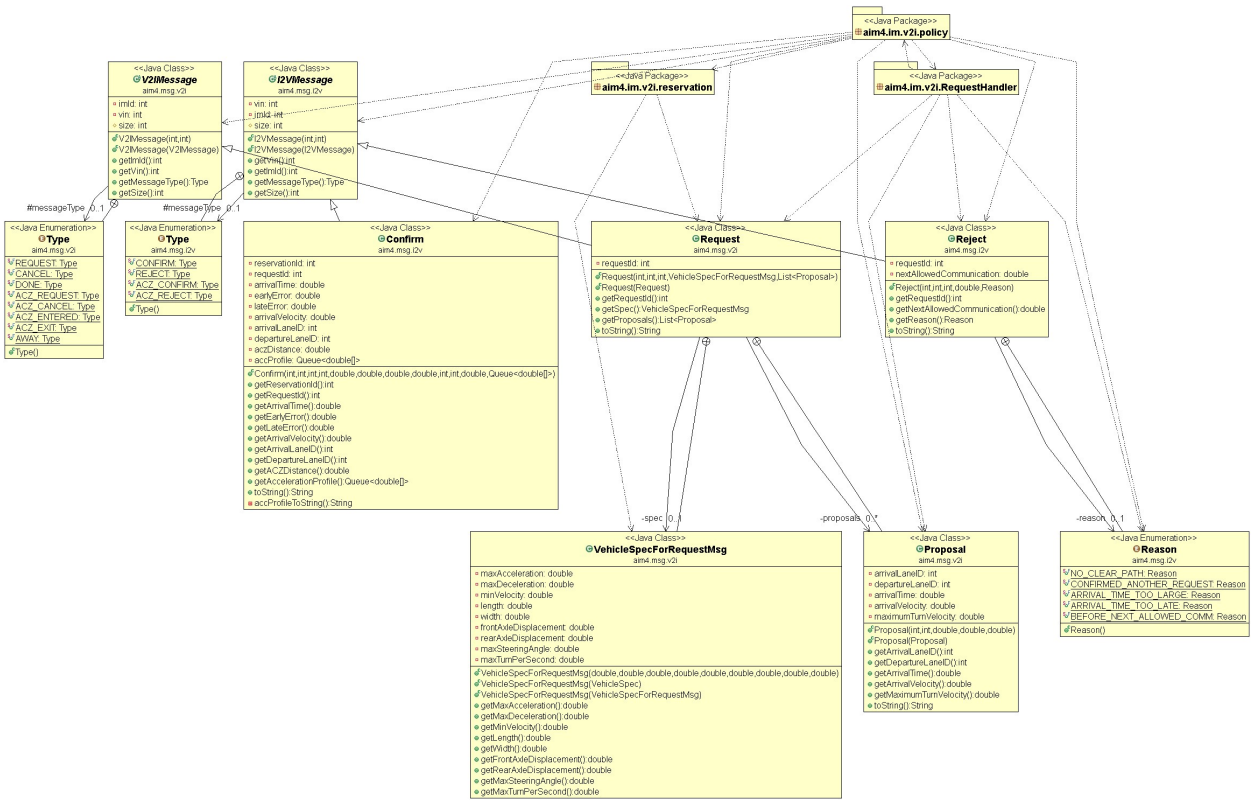


Figure 5: Reserve Relationship Diagram

Package Descriptions Overveiw

Below you will find a list of packages and classes found in the AIM5 code. Note, not all classes from the packages are named, just important ones which aid the understanding of the system and the ones which were changed.

Package (description)

- Notable sub classes/ Changed sub classes (Description)

aim4

- Main.java

aim4.config (Configuration of the simulation)

- Constants.java

- aim4.driver (Methods to allow the driver to act independently)

Autodriver.java

- Driver.java (Steps to show how the driver acts)

aim4.driver.coordinator (Methods on acceleration, arrival time, and reservation checking)

aim4.driver.navigator (Agent that chooses which way a vehicle should go)

aim4.driver.pilot (Controls how the driver "drives")

aim4.gui

- Viewer.java (What the user sees)
- Canvas.java (How the GUI looks)
- SimSetupPanel.java (Sets up the simulation according to inputs given in the starting page)

aim4.gui.component (Contains the java components used by the GUI)

aim4.gui.frame (Contains the java frames used by the GUI)

aim4.gui.parampanel (Contains the set up do the different parameter panel according to weather traffic lights or FCFS was selected.)

aim4.gui.statuspanel (Contains all the tabs from the GUI and their buttons)

- AdminControlPanel.java
- ConsolePanel.java
- PedestrainButtonPanel.java
- VehicleInfoPanel.java

aim4.im (Intersection Manager)

- Intersection.java (Interface of an intersection)
- IntersectionManager.java (Sets up the properties of the intersection)

aim4.im.v2i (Vehicle to Intersection AIM)

- V2IManager.java (Uses the policy in place to handle the intersection and to ensure no collisions)

aim4.im.v2i.batch

aim4.im.v2i.policy (Implementation of various intersection control policies for the V2I IM)

- AllStopPolicy.java (All vehicles stop at the intersection)
- BasePolicy.java
- TimeoutPolicy.java

aim4.im.v2i.RequestHandler (Handles requests from vehicle to intersection)

- AllStopRequestHandler.java
- FCFSRequestHandler.java
- GoStraightRequestHandler.java (Default)
- PedestrainsRequestHandler.java (Handles vehicle requests while pedestrian policy on, automatically on when pedestrian level above 0.)
- RequestHandler.java

aim4.im.v2i.reservation

- AdmissionControlZone.java (Only allows vehicles into intersection when there is room ACZ)
- ReservationGrid.java

- ReservationManager.java (Creates queries)

aim4.map

- DataCollectionLine.java (When vehicles cross a data collection line their statistics are collected and can be "dumped" by the user to analyze them)
- GridMap.java
- Road.java (Group of lanes with a name)
- SpawnPoint.java (point on map where the vehicles spawn)

aim4.map.destination

aim4.map.lane (Lanes created with separate IDs)

aim4.map.track (Determines the track of the vehicle)

aim4.msg.i2v (Messages from the intersection to the vehicle)

- Confirm.java (Vehicle path is safe to travel)
- Reject.java (Vehicle path is unsafe at this time and must submit another request)

aim4.msg.udp (For messages sent over UDP connection)

aim4.msg.v2i (Types of messages from the vehicle to intersection)

- Away.java (Message to state the vehicle has exited the admission control zone (ACZ))
- Cancel.java (Message sent to cancel and reservation)
- Done.java (Message sent when completing a reservation)
- Request.java (Message sent to request a reservation)
- V2IMessage.java (Message sent from vehicle to intersection)

aim4.noise (Contains functions to create noise)

aim4.sim

- AutoDriverOnlySimulator.java
- Simulator.java

aim4.sim.setup (Sets up the simulation with inputs given)

- BasicSimSetup.java
- AutoDriverOnlySimSetup.java
- SimFactory.java (Creates a simulator)
- SimSetup.java (Sets original values in simulator)

aim4.util (This class provides helper methods that are used throughout the code.)

aim4.vehicle (Vehicle specifications)

BasicVehicle.java

Data Structures Used

BasePolicy.java

Each proposal for a path across the intersection, made by a vehicle, is either rejected or accepted. These proposals are stored in a list. Each request a vehicle makes contains multiple proposals for the vehicle. A new linked list of proposals are copied and created to maintain data, they are filtered before iterated through by a filter that removes ones with times that will not work.

A mapping from VIN numbers to reservation Id is created through the creation of a vinToReservationId Hashmap. this allows the individual vehicle and its reservation to be found easily.

aim4.im.v2i.reservation

The Map of the road is run on a grid that contains tiles. The grid is a treemap containing integers. Treemapping is used for displaying hierarchical data and this is necessary in the grid as roads and vehicles and pedestrians needs to be represented.

RoadBasedIntersection.java

Roads and lanes are contained in array-lists to search through when checking if a path is clear. (entryRoads, exitRoads, roads, lanes)

PedestrainRequestHandler.java

An arraylist of integers is kept for tracking. This is to keep a list of the vin numbers of the vehicles in the simulation who are using the lanes.

Program Validation and Verification

This project is successful in the way it was implemented, as we have a working FCFS Autonomous vehicle intersection manager with pedestrian functionality. The team worked in an iterative style with testing at every step and committing to github when a step was completed.

We have included buttons to simulate the pedestrians pushing buttons to monitor the flow of traffic when pedestrians act. This is because pedestrians spawn randomly therefore to test we need exact input to monitor and analyze. A way of testing if the pedestrian buttons work is to start the simulation and pause it once a vehicle has spawned. If the vehicle is clicked on, its direction it is heading in will appear along with other statistics in a pop up box. The pedestrian lights can be turned on to monitor the vehicles interaction with the pedestrian light and if it stops or turns correctly. The vehicles path can be obstructed by anther vehicle, a road back up or a pedestrian walkway being lit up. The vehicle should slow down to avoid another car and stop to wait until safe to drive if there are pedestrians in the road. This is a way we tested and debugged the code. It is effective as the program can be paused at any point and stepped through slowly to be sure that it is performing as needed.

These test cases are written for a white box testing as we understand the internal structure of the environment. We did not include user tests as this was a functioning program before and our user manual has in depth instructions on how to run the simulator. A full list of test cases and results can be found in appendix B. A summary can be found below.

Table 1: Summary Testing Plan

Process	**Technique**
GUI Testing: test the interaction of the sliders and buttons on the simulation	Random and edge case testing
Interactive Class Testing: testing each class's new features merge successfully with old	White box and Behavioral Testing
Validation Testing: test whether customer requirements are satisfied	Use-case based black box Tests
System Testing: Run the simulation in full	Stress and performance tests
Pedestrian Testing: Test if pedestrians can control buttons and move	White- box and behavioral testing
Vehicle Testing: Test if vehicles can pass through intersection without collisions	White- box and behavioral testing
Optimization Testing: Comparative testing between original AIM4 and new AIM5	Stress and Performance tests

Summary of Results

After running the test results it can be seen that our program runs as wanted and cars react to pedestrian lights via the intersection manager smoothly. The optimization of the code could be better as the AIM5 does not cope as effectively over the same time for as throughput as AIM4. However Aim5 has a larger breaking point which could be down to better lane throughput when backed up traffic occurs.

Conclusion

We managed to provide an autonomous vehicle intersection manager that runs successfully without crashing and while adding extra functions. The way in which we implemented pedestrians was successful as the paths they travel is very visible and vehicles that do not follow the prescribed policy would be easily seen traveling over the line. However, vehicles listen to the policy and stop and wait for pedestrians to move. Pedestrians spawn and decide where to walk and push the button to start a timer which manages the flow of pedestrians.

We optimized the code as can be seen by the threading used in the vehicle classes. We optimized the way in which vehicles stop to wait for pedestrians if they can by calculating if it is possible beforehand. The optimization of the code can be seen by the increased breaking point but when compared the new program does not preform as optimally as the original. This could be because of the added functionality or the small scale for using threads which is not utilizing them to their best ability.

While understanding and working on the code we discovered that working with multiple classes was a large job and future developers using this code should try to decrease the number of classes if possible. We attempted to, but felt like our time was more useful in the development of new features and not reduction of old. Our program is suited to large scale use as the threading will increase efficiency and suggest to future developers to continue to add features. Features we would have liked to develop would be human driven cars, and a more optimal algorithm. The pedestrian Class could be extend to show moving people on the GUI.

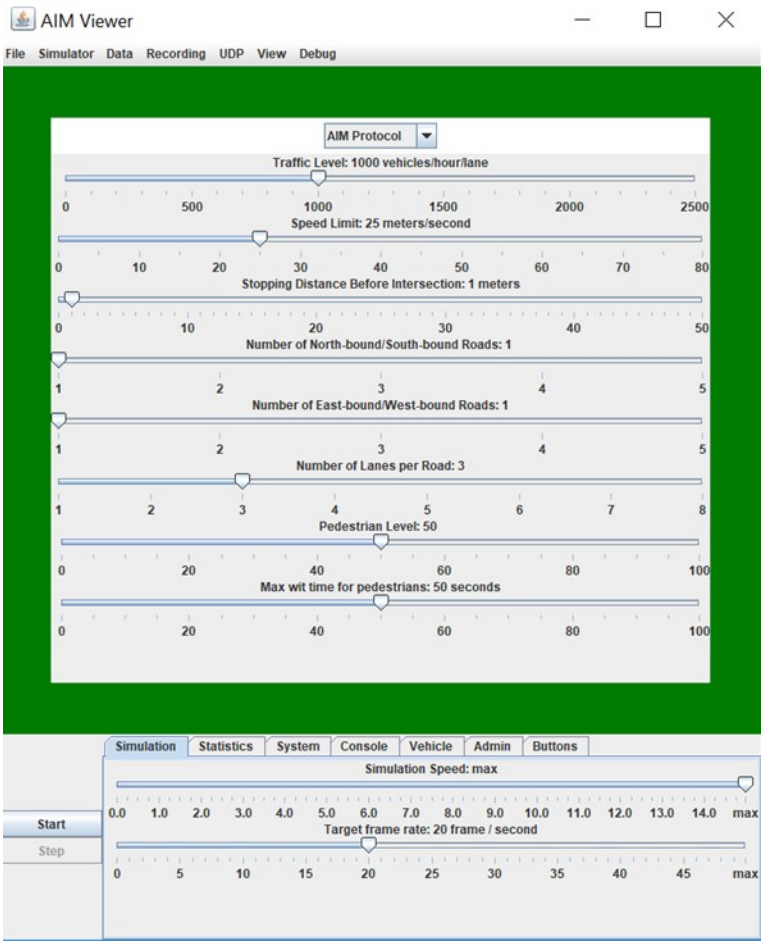


Figure 6: Starting Screen with Default Values

Appendix A

User Manual

This is a manual on how to operate the AIM Simulator.
Open simulator to starting screen.

Starting Screen:

Choose AIM, Traffic signals or stop signs as your protocol of choice.
If AIM selected:

- Set your desired traffic level on the traffic level slider.
- Set the speed limit.
- Set the number of roads north, south, east and west bound.
- Set the number of lanes per road.
- Set the pedestrian level. (If above 0, pedestrians request handler policy is activated)
- Set the maximum wait time for pedestrians.

If Traffic Signals selected:

- Set your desired traffic level on the traffic level slider.
- Set the number of lanes per road.

- Set the green and yellow signal duration times.

If Stop signs selected:

- Nothing to set.

Tabs found at bottom of the screen:

Simulation:

- Set your simulation speed and your target frame rate, current time, completed vehicles, average data transmitted and average data received.

Statistics:

- Gives a summary of statistics while simulation is running.

System:

- Gives summary system information such as, Operating system, Java version, memory usage, allocated JVM memory, maximum JVM memory.

Vehicle:

- When clicked on a vehicle in the simulator, vehicle information is shown such as, VIN, vehicle type, velocity, acceleration, data transmitted and data received.

Admin:

- Contains buttons linked to different policies that can be enforced, First come first served, all vehicles to stop, Alternating vehicles and the pedestrian policy. To enforce a policy, the intersection must be clicked on and the chose the policy.

Buttons:

- Buttons used for testing the pedestrian lights to stop the vehicles. Each button likes to a pedestrian path on the intersection, bottom, top, left, right, top right to bottom left, top left to bottom right.

Running the Simulation

Click the Start button to begin the simulation.
The timer at the top right hand corner of the screen will start counting up. The initial simulation time is 0 and it increases at an increment of 0.02 second by default to show faster than normal life speed.
Vehicles will begin to spawn and cross the intersection.
Vehicles change colours as the move along the road.

- White- has a confirmed reservation and can enter intersection
- Blue- sent request to intersection manager but has not gotten a reply
- Yellow- driving default, have not yet received a reservation and are not currently waiting for a reservation

Pedestrians will begin to spawn. There are three white blocks on each corner of the intersection and when a pedestrian spawns and wants to go in a particular direction the block will change to pink to indicate pedestrians are spawned there.

To see system information on a specific vehicle click on it while it is moving to activate a pop up box. Information can also be found in the Vehicle tab on the bottom panel.

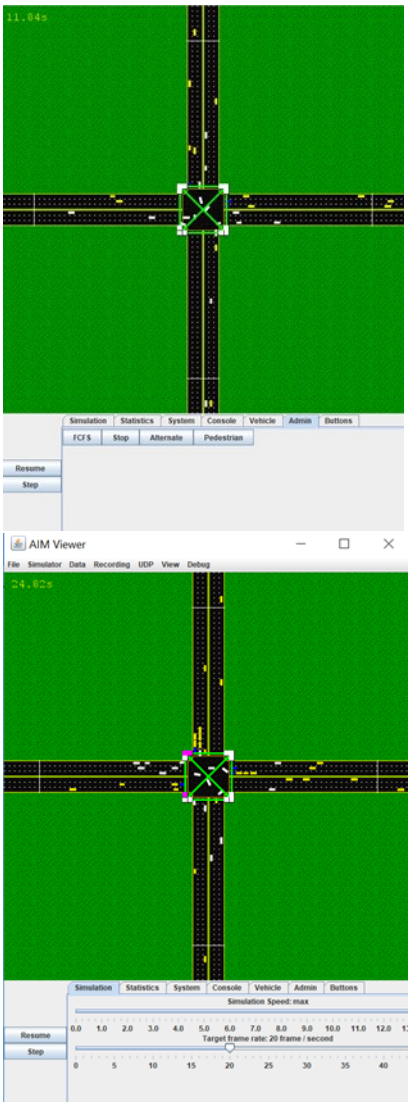
Pause the simulation at any point using the pause button and use the resume button to resume.

Once paused you can step through the system with the Step button to see each individual step.

To use the buttons to simulate pedestrian control, click in the intersection wanted, select pedestrian in the Admin tab and then the pedestrians will affect the IM. In the buttons tab, select the path wanted to turn the lights on.

Once turned on a yellow line will appear on the line the pedestrian would walk on. After the max wait time has elapsed the line will change to red and pedestrians will cross and cars will stop.

The white lines near the beginning and the end of the roads in the map are the data collection lines, which are used to collect the data about the vehicles that pass over them. The VIN of the vehicle, in conjunction with the time at which the vehicle crossing the line, is recorded and this information stored for the simulation.



Appendix B

Test Cases

Table 2: Testing the Whole Program

Test Explanation	Expected Result	Result
1. User loads simulator and runs it.	The startup page should show sliders and the main page should show an intersection with cars spawning.The GUI does not crash.	Simulator runs and does not crash
2. Set speed and vehicle traffic levels to maximum on AIM4 and AIM5, run both and compare breaking point*	Both should display the same breaking point of traffic congestion	AIM5 69.50, AIM4 62.50
3. Run both simulations one after the other with the default inputs and no pedestrians. Run for 40 Seconds and compare throughput.	The simulators should appear to function similarly although they will not be identical due to random spawning of vehicles and their paths.	AIM5: 56 cars, 52 cars, 47 (slow computer), AIM4 58 cars, 57 cars, 62 cars. Therefore AIM5 not as optimal for throughput over 40 seconds
4. Run both simulations one after the other with the default inputs and pedestrians. Run for 40 Seconds and compare throughput.	Pedestrians will reduce throughput but should not hinder much.	Aim5: 20 cars, 7 ped, ave wait time 5.77.37 cras, 3 ped, ave wait time 13.56. AIM4 56 cars, 57 cars, 59 cars.
5. Run program on Windows and Apple machine	Both should run identically	Both run the same.

*Breaking point- when at least one lane become backed up till the end of the lane to the data collection point for each direction

Table 3: Testing Sliders

This it to test the starting options sliders. The figures selected should display/be visible in the simulation when running. These sliders are important for the integrity of the simulation and without being able to change the variables, the simulation becomes less lifelike and fails to simulate real world activities.

Test Explanation	Expected Result	Result
1. Set your desired traffic level on the traffic level slider to 0. Leave other variables. Press start.	No vehicles should appear.	No vehicles appeared, time continued
2. Set your desired traffic level on the traffic level slider to above 0. Leave other variables. Press start.	Vehicles should appear.	Vehicles spawn and drive along road.
3. Set the speed limit to 5 and observe slow moving vehicles.	Vehicles move slowly	Slow vehicles
4. Set the speed limit to 80 and observe fast moving vehicles.	Vehicles move quickly	Faster vehicles

Test Explanation	Expected Result	Result
5. Set the number of roads north, south, east and west bound to 1.	Observe a 4 way stop.	4 way stop created
6. Set the number of roads north, south, east and west bound to 2.	Observe 4 intersections.	4 Intersections created
7. Set the number of lanes per road to 1.	See one lane per road.	1 lane per road
8. Set the pedestrian level to zero. Select intersection and set to pedestrian policy.	No pedestrians should be seen.	No pedestrians
9. Set the pedestrian level to 50. Select intersection and set to pedestrian policy.	Pedestrians should be seen by white blocks becoming pink	Blocks become pink from white

Table 4: Testing Pedestrians and Lights

Test Explanation	Expected Result	Result
1. Select the right/left/top/bottom button.	Traffic going to and from that direction should stop. Vehicles using other lanes should continue.	Traffic stops and continues when button is pressed again.
2. Select the top left to bottom right or top right to bottom left.	Cars should react appropriately(see Car turning diagram below). Cars closest to the lane they want to turn into should be able to but no cars should turn over the path that pedestrians would be crossing on.	Cars react appropriately and do not cross where pedestrians are
3. Pedestrians push button	Yellow line for waiting shows on desired route	Yellow line appears
4. Pedestrians cross	After line changes to red and pedestrians have crossed, line reverts	Line changes to red then disappears after pedestrians have crossed.

Table 5: Testing Vehicles

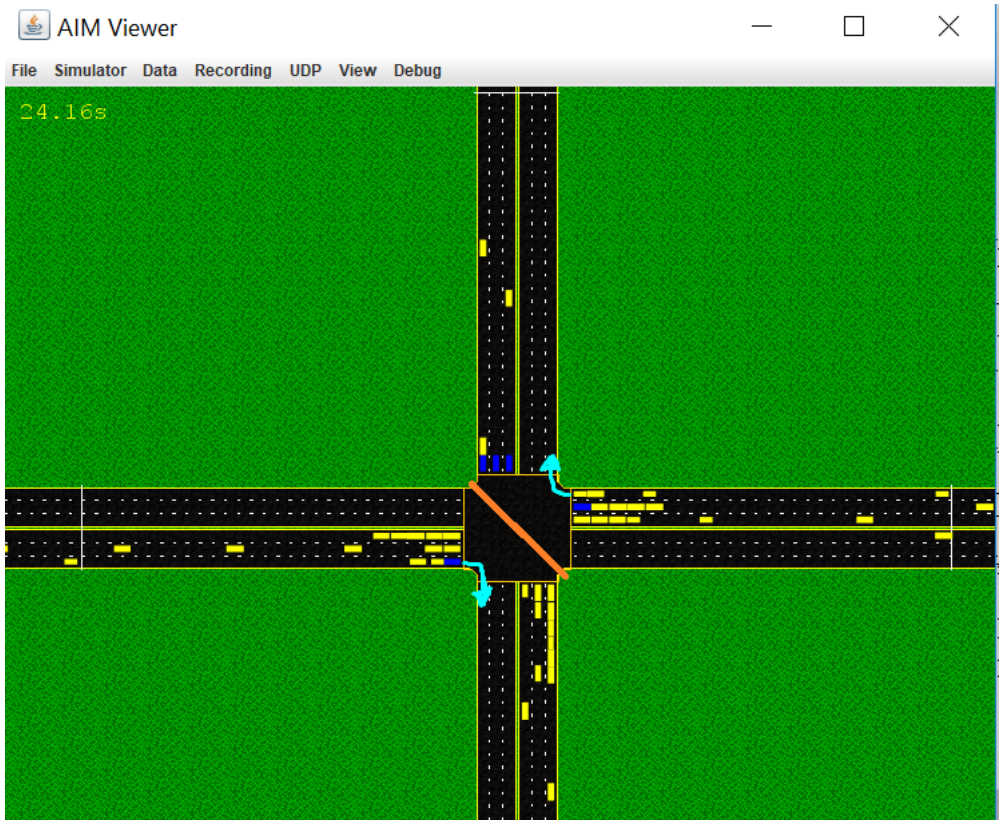


Figure 7: Car Turning Diagram

Test Explanation	Expected Result	Result
1. Click on a vehicle while moving	Vehicle information should be displayed	Vehicle information is displayed in pop up box

Appdendix C

References

Parker, A., and Nitschke, G. (2017). How to Best Automate Intersection Management. In Proceedings of the IEEE Congress on Evolutionary Computation, pages 1247-1254, IEEE Press.

Dresner, K., and Stone, P. (2004). Multiagent Traffic Management: A Reservation-Based Intersection Control Mechanism. In Third International Joint Conference on Autonomous Agents and Multiagent Systems, pages 530-537, ACM.

R Core Team (2013). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. URL HTTP://WWW.R-project.org/.

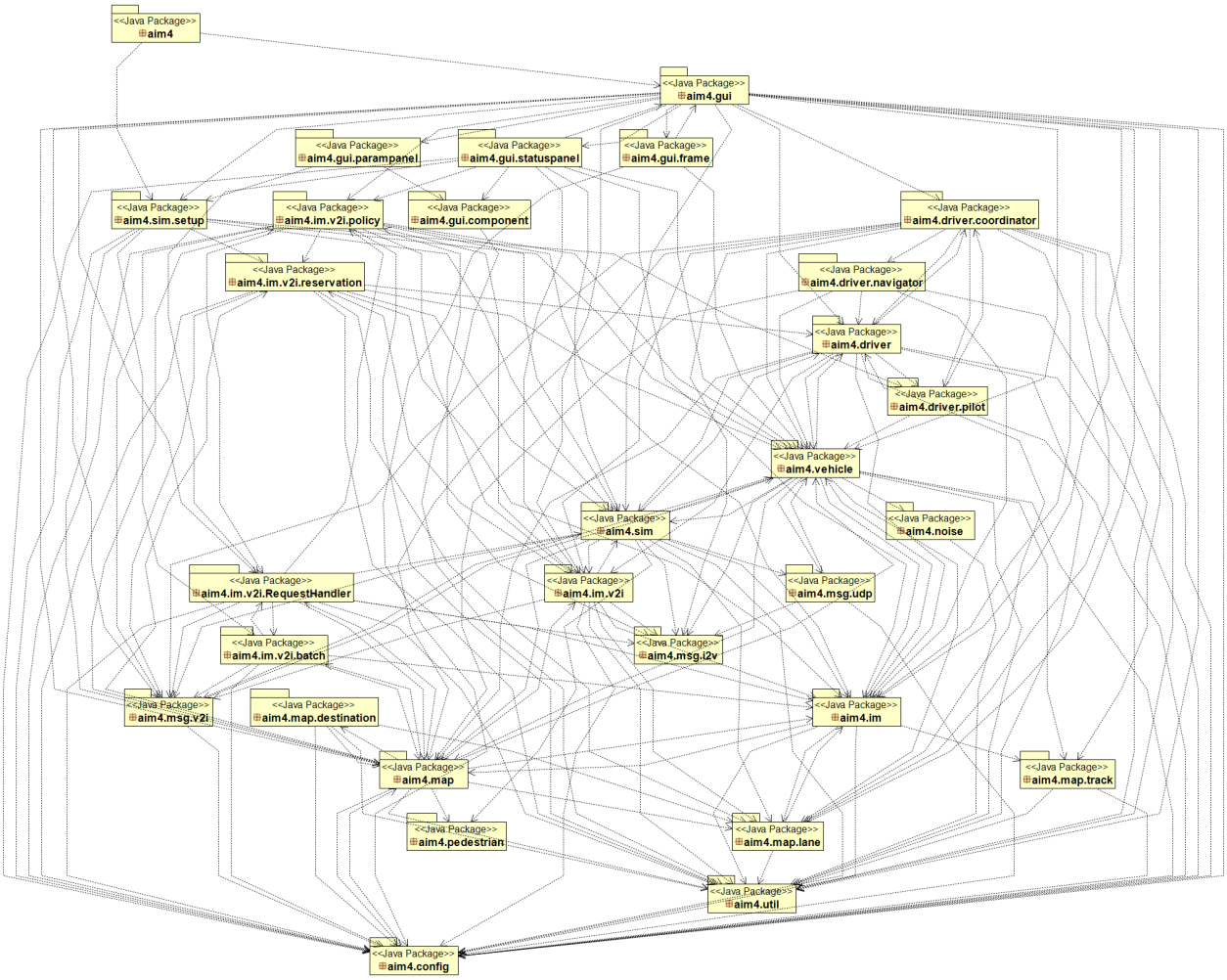


Figure 8: Complete Package Diagram

Plagarism Statement

I know that plagiarism is wrong. Plagiarism is to use another's work and pretend that it is one's own. I have not allowed, and will not allow, anyone to copy my work with the intention of passing it off as his or her own work.

Signature Jordan Jooste, Robbie Perrot, Jonathon Weideman
Date 06/09/2018