Summary

The purpose of this project was to create a computer simulation of an intersection without stop signs or traffic lights and develop artificial intelligence capable of making decisions autonomously. This intersection would be more efficient in terms of letting more cars through the intersection per minute, and also by eliminating the stop-and-go traffic flow of a normal, controlled intersection. This simulation was created in the Microsoft Visual Studio 2017 IDE using C#, an object oriented programming language created by Microsoft geared towards game design, and leveraged the MonoGame framework, an improvement to Microsoft’s XNA. The physics and mathematical details of the simulation were created by the experimenter and derived from data from real world tests on the Tesla Model S. Compared to last year's project, much more time was allocated to the artificial intelligence and expandability of the experiment, meaning the simulation could easily model multiple real world intersection types, with the artificial intelligence continuing to function well. Vehicles, unlike last year, are able to turn and navigate the intersection in a much more realistic manner and collisions are detected at a visibly higher rate. Based on hours of testing, running, debugging, and revising the simulation, the observed results support the hypothesis. The cars noticeably detect future collisions and make decisions either to stop, slow down, or speed up, such that collisions are avoided.

Abstract

The purpose of this project was to create a computer simulation of an intersection with no traffic controls (stop signs or traffic lights), and develop artificial intelligence capable of making decisions autonomously, meaning self-driving cars regulate their speed and pass through the intersection without collision. This intersection would be more efficient in terms of letting more cars through the intersection per minute, and also by eliminating the stop-and-go traffic flow of a normal, controlled intersection.

This simulation was created in the Microsoft Visual Studio 2017 IDE using C#, an object oriented programming language created by Microsoft geared towards game design, and leveraged the MonoGame framework, an improvement to Microsoft's XNA. The physics and mathematical details of the simulation were created by the experimenters specifically for this simulation. Acceleration and deceleration physics were derived from data from real world tests on the Tesla Model S.

Due to the progress made in developing the simulation last year, less time needed to be spent on this aspect of the project, but still required attention to improve its physics and overall ergonomics. This allowed more time to be spent on the artificial intelligence and expandability of the experiment, meaning the simulation could easily model multiple real world intersection types, with the artificial intelligence continuing to function well. Vehicles, unlike last year, are able to turn and navigate the intersection in a much more realistic manner and collisions are predicted and detected at a visibly higher rate. Based on hours of testing, running, debugging, and revising the simulation, the observed results support the hypothesis. The cars noticeably detect future collisions and make decisions either to stop, slow down, or speed up, such that collisions are avoided.

Autonomous Intersection:

Developing and Simulating Self-Driving Car A.I. Without Need for Traffic Lights

Introduction

The arrival of fully self-driving cars is inevitable within the next few years, and for years this idea has peaked our interest. After spending too many hours of our lives sitting at intersections and stopping at stop signs with no one in within 100 meters of the car we were traveling in, we wondered if self-driving cars could be leveraged to eliminate this wasted time. By having the self-driving cars communicate their distance from an intersection they could regulate their speed allowing cars to pass through the intersection without unnecessary stopping, and more importantly, not crashing.

The purpose of this project is to improve the computer simulation of this intersection utilized last year, and develop artificial intelligence capable of making decisions autonomously. This simulation will be made using some work done last year, while still drastically improving its functionality. Additionally, the basic principles of self-driving cars had to be created from scratch, as the artificial intelligence from the year previous has been completely removed. This simulation will use a fully, highly improved decentralized system, where each car makes decisions for itself instead of being controlled by a central hub. While this adds some complexities and redundancies, this type of system is more reliable and resistant to hacking or failures if implemented in a the real-world situation. This simulation will use a traditional four-way intersection, rather than a roundabout, even though a traditional intersection is less efficient and makes implementing this idea harder. Use of a traditional and widely available intersection is important to the simulation to prove that this idea can be achieved with existing infrastructure. As opposed to last year, this four way intersection will be expandable, making its implementation much more feasible in a variety of intersections.

Review of Literature

A real-world object is defined by two main characteristics: its state and behaviors (“Object-Oriented”). The state of an object is the combination of its qualitative and quantitative characteristics (“Object-Oriented”). The behaviors are the actions that can either be done to or by the object. Object oriented programming takes this concept of objects and provides means of defining an object in code through its state and behaviors (“Object-Oriented”).

Object Oriented Programming (OOP or OO) is a design philosophy where code is grouped into self-sustained modules, called classes. A class is the blueprint or template for a type of object that describes the fields and methods available to the object. Therefore, an object, in its truest sense, is just a specific instance of a class (L.W.C). In object oriented programming, the state of an object is defined by fields, usually in the form of variables. The behaviors, however, are defined by methods, which are executable functions that operate on the object’s state and are executed by the outside world, or by the object itself (“Object-Oriented”). All methods, on the most basic level, are either get or set functions, where a get function retrieves the value of fields and a set function modifies the values of fields.

The functions in a class which take parameters used to create new instances of the class are called constructors (L.W.C). To create (declare) a new object, a *new* instance of the class is declared by calling the constructor function, with a corresponding name, to create a new object instance of that class. If a class has a constructor function, then the object can be declared using the constructor, which usually includes parameters, used to modify the object's fields. Alternatively, using no constructor would just create a default instance of the class, with all the default values defined in the beginning of the class file.

To reduce complexity and help manage class files of a project in object oriented programming, encapsulation, inheritance, and polymorphism are used (L.W.C). Inheritance is "the ability of a new class to be created from an existing class, by extending it" (L.W.C). This feature is useful for defining more specific subclasses that have unique properties or methods but all share basic properties and methods from a base class. Polymorphism can be understood as the ability to call the same operation or method but requesting a wide range of functionality (L.W.C). Falling under the category of polymorphism, method overloading is an extremely useful feature of object oriented programming. Method overloading is the ability to define multiple methods all under the same name, but with different quantities or types of parameters to distinguish them (L.W.C). This can be useful for having multiple ways to declare a new object or to have different versions of an action or behavior. These three methods are useful for programmers for reducing complexity of programming, and are just one set of advantages to object oriented programming.

As compared to standard procedural programming, object oriented programming has several distinct advantages, including modularity, data encapsulation, reuse of code for other projects, and simpler debugging of broken parts ("Object-Oriented"). Modularity refers to the ability to make several independent pieces of software and then linking them into one system. Reusability is highly related to modularity, and comes from the fact that classes are stand alone pieces of software, as long as they do not rely on code inherited from parent classes. Also thanks to modularity, debugging is simpler because each module of code can be debugged and fixed individually, rather than trying to debug and fix an entire file of code (“Object-Oriented”). The purpose of data encapsulation is “to hide how a class does its business, while allowing other classes to make requests of it”. This is is achieved by making a classes’ properties and methods public (L.W.C). Public methods and fields are accessed by calling the name of the object, followed by a period, and then by the exact name of the desired method or field.

“C# is a modern object oriented programming language developed in 2000 by Anders Hejlsberg at Microsoft as a rival to Java” (Mkhitaryan). It is the fourth most popular programming language, and 31 percent of all developers use it regularly. “C# is often thought of as a hybrid that takes the best of C and C++ to create a truly modernized language” (Mkhitaryan). Syntax-wise, it is also extremely similar to Java, but for aspects of object oriented programming such types, objects, and more advanced structures and features, C# is more elegant and simply better: “The C# build process is simple compared to C and C++ and more flexible than in Java” (Wagner). C# improves on Java by providing features such as nullable value types, lambda expressions, and direct memory access (Wagner). Unlike C and C++, no header files are needed in C# which makes it even simpler.

C# can be used to create almost anything but is particularly strong at building Windows desktop applications and games (Mkhitaryan). For game creation, C# commonly leverages Unity, the most popular game engine. An advantage of C# is that complex tasks abstracted away, so the programmer does not have to worry about them, unlike Java (Mkhitaryan). For example, .NET's garbage collection scheme handles memory management for the user, so the user can focus on programming. Since C# is a statistically-typed language, the code is checked for errors as it is typed out, constantly feeding back error messages, making debugging and error management much simpler (Mkhitaryan).

Microsoft's .NET Framework is "an integral component of Windows that includes a virtual execution system called the common language runtime (CLR) and a unified set of class libraries" (Wagner). The .NET framework simplifies compiling and assembly for multiple languages. Additionally, it includes an extensive and feature packed library of over 4,000 classes (Wagner). These classes are organized into namespaces which make accessing the libraries in code intuitive and straightforward. The libraries included in the .NET Framework make coding simpler by eliminating the need to do tedious or long tasks manually. The C# programming language relies on the .NET Framework to run (Wagner).

The XNA Framework is a set of tools provided by Microsoft meant to facilitate game development ("XNA Frequently Asked Questions"). Humorously, XNA is an acronym for "XNA is Not an Acronym". XNA is an expansion of the .NET Framework with libraries specifically chosen to help develop lightweight games that can be run on a variety of Microsoft platforms. XNA's release was first announced on March 24th, 2004, and on January 31st, 2013, after nearly a decade, Microsoft announced that XNA would no longer be developed ("XNA Frequently Asked Questions"). Therefore, an open source cross platform version of XNA, called MonoGame, was developed to replace it.

Development of MonoGame began in 2009 by José Antonio Leal de Farias, an active member of the XNA community ("About"). Originally known as XNA Touch, it was later renamed in 2011 to MonoGame. The goal of Monogame is to allow XNA developers on Windows platforms to port their games to non-Windows platforms, and to also have an open source game development library ("About").

"Swarm intelligence is the discipline that deals with natural and artificial systems composed of many individuals that coordinate using decentralized control and self-organization" (Dorigo). Swarm systems studied in nature include ant colonies, schools of fish, and flocks of birds. By studying how the individuals in these swarms communicate, scientists hope to accomplish one of two things: mimic these interactions, or extract ideas for application elsewhere (Miller). A highly prevalent application of swarm intelligence is in swarm robotics, where swarm intelligence is incorporated into the artificial intelligence of a system of robots (Dorigo). This has proven to be very difficult due to the challenge of creating successful code that can overcome both hardware and sensor limitations.

“The characterizing property of a swarm intelligence system is its ability to act in a coordinated way without the presence of a coordinator or of an external controller” (Dorigo). This description of swarm intelligence can be broken down into three main properties. First, the system must be composed of many individuals, to actually be a swarm. Secondly, the individuals should be generally homogenous, or similar in type. Finally, the interactions among individuals are based on simple behavioral rules (Dorigo). This final property will be more difficult to achieve when trying to make decisions for a self-driving car, which has many different factors to weigh in the decision-making process.

The properties of a swarm system previously discussed allow them to be designed to be scalable, parallel, and fault tolerant (Dorigo). Scalability means that a system can maintain its function as the number of members increases without needing to modify the way the members interact. This is a key feature, and challenge, of the autonomous intersection. Parallel action is the ability for members of a system to execute different actions in distinct places all at the same time (Dorigo). “Fault tolerance is an inherent property of swarm intelligence systems due to the decentralized, self-organized nature of their control structures” (Dorigo). For swarm systems in nature this is true, but for an intersection of cars, fault tolerance has to be a designed feature. There is, however, inherit fault tolerance for the intersection of cars because it will be a decentralized system, making it less prone to hacking as compared to a system controlled by one centralized computer or server.

Collections in C# programming language are data storage and retrieval classes that include lists, hash tables, stacks, etc. (“C# Collections). Arraylists are a type of collection and allow users to append and delete items from anywhere within the class, with the data structure resizing itself automatically (“C# Collections). These values are accessed via an index and, unlike normal arrays, do not have a fixed length and are incredibly useful in situations where expandability is important.

Hypothesis

If the artificial intelligence is properly created using swarm intelligence, then the self-driving cars in this computer simulation will regulate their speeds and make decisions autonomously, such that no traffic lights are needed.

Materials

- Computer With Windows OS Installed
- Log Book
- Graph paper notebook
- Pencil
- Pen

Procedure

The following is a brief summary of the procedure of this project.

- Evaluate environment used previously to create simulation in. Review the basics of that language.
- Learn how to code sprites and their movements and rotations.
- Learn object oriented programming.
- Research acceleration and braking physics for cars.
- Make all core values adjustable in a file in the program.
- Add scale to program and then test to make sure the scale is accurate.
- Draw out the road based on specifications stored in variables.
- Make cars be able to turn, unlike last year. This is much more complicated than it seems.
- Spawn in multiple cars. Make cars spawn in and terminate for an infinite amount of time.
- Create and implement simple self-driving car artificial intelligence, such as braking and accelerating based on the car ahead.
- Make cars detect and avoid obvious and less apparent collisions
- Have cars react and weigh multiple potential collisions, avoiding these ultimately.
- Add acceleration logic.
- Make cars queue up at intersection and unqueue efficiently.

- Add basic logic that humans innately have, such as not accelerating when a car is stopped directly in front of them.
- Add keyboard controls to make troubleshooting easier.

The very first step of this project was to evaluate the environment and programming language used last year to see if its use did not harm the development of the project. Eventually XNA was again determined to be a good fit for the project and would allow for drastic improvements from last year. Microsoft Visual Studio IDE 2017, along with XNA 4.0, and MonoGame pipeline were used in developing this version of the project.

C#’s syntax is nearly identical to Java, making learning it relatively easy to learn for the new member to the group. Therefore, the next step was to learn/review the basics of C#, and become confident in the syntax, and how to implement things such as matrices and for loops. After becoming proficient in procedural programming in C#, it came time to learn how to code the visual elements of the simulation using XNA and MonoGame. “RB Whitaker’s Wiki” provided great resources for learning XNA, and also MonoGame. Once this was accomplished, the project officially began.

Acceleration of the cars had to be improved, as in the year prior cars were jittery and did not reflect reality. The overall acceleration physics did not change, but the way in which the cars utilized these values did. Throttle target was the first notable improvement, which allowed vehicles to seek a desired acceleration value.

Throttle time and throttle applied were added to allow cars to accelerate with much more variability. This meant a braking or accelerating value could be applied for a set amount of time, limiting constant, inconsistent changes in velocity, thereby improving the simulations ability to reflect reality. The possible acceleration based on the vehicles current velocity was revisited and determined to be accurate based on the Tesla Model S values used to originally calculate this value.

Braking physics improved accordingly, with the same principles used to improve acceleration consistency present. Idling deceleration, meaning cars slow down when there is no gas being applied to the car, was revisited and determined to adequately account for both friction on the axles and on air resistance. The friction calculation is dependent on the weight of the car; the air resistance calculation uses accurate wind tunnel data and aerodynamic properties of the Tesla Model S.

Having global variables makes adjusting core aspects of the simulation very simple and error proof, as compared to changing every instance of a value throughout several files. This was accomplished by creating a single class file that contained all the core variables and specifications of the simulation. Public variables were used to make them accessible by all files throughout the simulation.

To make the simulation accurate and realistic, as well as consistent throughout, a scale needed to be established. The scale is in pixel per meter. At first there was going to be one fixed scale, but then by storing the scale in a global variable, and referencing that variable, an adjustable scale was created. Then to test if the scale was accurate, a car was driven across the screen at a set speed, the time it took to go across was recorded with a stopwatch, and compared to the calculated expected time, based on the width of the screen and speed of the car. This was done again to ensure the accuracy of the previous calculation.

Then the world had to be created. Rather than draw out the world in a paint program, the world would be drawn out by the simulation. This allows the world to be resized and adjusted by just simply adjusting variable values. The variables used to draw the world include the number of lanes, width of lanes (in meters), road line thickness, space between dashed road lines, and more. This was unchanged from last year.

The archived turning code did not function, meaning turning had to be redesigned. To get cars to turn, five values have to be known about the car. It's speed, desired turning radius, direction, turning point, and it's current angle or rotation. The speed is used to determine how much distance would be covered by the car on a given time interval. Then the turn radius is used to find the total turn distance (a quarter of a circle's circumference). The distance traveled by the car is then taken as a percent of the total turn distance. That percent is then multiplied by 90 degrees and added to the car's current rotation. This new angle in degrees is then converted to radians. The cosine and sine values of this angle are used to find the new x and y values of the car, by adding them to the car's turning point (the center of the turning radius' circle). A different method is used to turn for each direction, and for left and right turns. The turning functions are still included in the code and work flawlessly and, unlike last year, are utilized within the simulation.

Spawning vehicles was redesigned as well to improve efficiency and ease of access. To spawn in multiple cars, a list, rather than an array like last year, of car objects are initialized once at the beginning, and then at random intervals, the car objects in that array are replaced sequentially. And once the end of the list is reached, it loops back to the beginning if the first car in the array is not active. The car's active variable is a boolean that is set to true when it is spawned in, but is inactive before being spawned in, and after being terminated, which occurs after the cars goes off the screen.

Collision detection is very resource intensive, so an efficient yet accurate method of collision detection needed to be created and implemented. And unfortunately, there was no built-in collision detection code in XNA or Monogame. Several methods were drawn up on scratch paper, and the method decided on was deemed to be the most reliable, but uses extensive computing power. It detected if corners of the car's hitbox were in another car's hitbox. To keep the hitbox rotating with the car, the unit circle was leveraged. The hitbox consisted of four corners of the car, as well as midpoints on each side, totalling eight points. Once a car was determined to be within range of a possible collision (one car length), all of the corners were tested to see if they were within the other car's hitbox by finding the slope of each side of the car's hitbox and testing if the corner fell within the slope of each side. A simple collision code for "t-bone" collisions (when cars are perpendicular) was also added. This, however, is not used in the simulation as it requires too much processing power to calculate for every vehicle and merely observing the intersection in action proves whether or not cars collide.

After programming, creating, editing, and testing all the other parts of the simulation, it was finally time to redesign the artificial intelligence of the simulation. The first aspect that required redesign was speed management, meaning maintaining appropriate speeds based on the speeds of the cars around it. Based on the distance to the car ahead, utilizing the aforementioned lidar, and the speed of the car itself, cars were able to maintain an appropriate following distance. This was a notable change from last year as instead of calculating a following distance using the speeds of other cars and their distance to intersection along with a host of other values, cars now need very little information to maintain this distance. If the following distance is too short, cars will automatically brake to increase this distance, and when it becomes too far, cars can accelerate to close the gap. The necessary following distance changes as the car brakes, meaning as the car’s velocity decreases, the following distance necessary decreases until it hits its minimum, indicating a needed stop. This avoiding unnecessary consideration of other cars’ velocities.

The next necessary improvement was to vehicle intercommunication, as the previous method required cars to parse relevant information from a master array. To improve efficiency and practicality of implementing intercommunication into the real world, cars now push their states, a set list of relevant values like intersection id, vehicle velocity, vehicle acceleration, etc., to the “airwaves” list, which models transmission via broadband on designated frequencies. From there, vehicles copy this list to be used in their own data analysis. This drastically improved the efficiency and expandability of the project, as before cars had to scan through a complete list of every present vehicle and manually extract all necessary values. This information is then used to see which cars could potentially collide, and if they can, what the estimated arrival time to the point of collision would be.

Using the information gained from the state transmission, the most important steps in the artificial intelligence are done. Once a car is determined to be the front most car in the intersection, it scans these values to determine first relevant vehicles, or any vehicles also in the front of their respective lanes or in the intersection center itself. From there, based on the vehicles destination, or what direction it is headed, it appends vehicles to this list. Then it scans through this array to generate an important vehicle array, separated into four different lists including important vehicles from each direction and those in the intersection center. For example, if a car is turning left, it will append any vehicles coming from the opposite direction not turning into this array. This is incredibly important as from this array the vehicle determines both its eta to determined points in the intersection and the relevant vehicles eta to that same point to see if a collision is imminent. This technique is the bulk of the artificial intelligence, as by weighing the possibility of collisions based on eta points generated at the beginning of the simulation based on intersection type, cars can determine if it is safe to enter the intersection.

Keyboard controls were previously added to improve debugging of the simulation and remained the same. What was added was click functionality allowing the user to select a vehicle while it is driving to view important information and once it has exited the screen, these values disappear.

Analysis and Conclusion

Overall, the simulation supported the hypothesis, and accomplished the central goals of this project. The cars better predicted potential collisions and adjusted their speed to avoid them, and needed no traffic signals nor an external controller to make these decisions. Therefore, to additionally support the hypothesis, swarm intelligence was successfully implemented and to a much higher degree than the year prior..

The simulation worked excellently with the flaws from the year prior being almost completely eliminated. The important aspects were improved from last year and remained suitable for the task at hand. For instance, the drawing of the world worked perfectly, the physics are accurate, and the simulation’s scale is accurate. Although the physics are accurate, they are not overly complex. This can be seen in the acceleration of the car where friction is not explicitly a factor, but it is a factor because it affects the real world data used to derive the acceleration physics.

Unlike last year, much more time was dedicated to the actual artificial intelligence of the project rather than being tied up in simply troubleshooting the simulation itself. Many hours were spent pondering various ways to make this process as simple and expandable as possible, without becoming to concerned with the minute details present in many intersections in the world today. The goal was to make this artificial intelligence as widely applicable as possible and this seems to be the case upon the conclusion of this project. Lane number, width, and a variety of other factors can be altered with the intersection largely functioning in the same way. Instead of using an overly complex scoring system to evaluate collision probability, this year the focus was put on a highly functioning system that allowed for vehicles to turn, a drastic improvement in the abilities of the code. Vehicles determine potential collisions only with important, relevant vehicles and navigate accordingly, adjusting velocity and acceleration accordingly. This is an immense improvement from last year and results in a much more manageable and efficient intersection.

Another factor in this project’s improvement was the foregoing of implementing unnecessary functionality that would prove superfluous in the real world. The specific function this refers to is collision detection, as in the real world both the passengers and car itself would most likely be able to determine the vehicle crashed. This made running the simulation much easier as processing power was not tied up in calculating multiple hitboxes that would only prove necessary if the project itself failed.

Although this project improved on the work done in the year prior, it is not perfect. In the future, much more focus could be put on the scalability of the artificial intelligence for unusual intersections that do not follow traditional set ups. This project could also contain vehicles of different sizes and capabilities, such as trucks and cars of different sizes, to improve the real world capabilities of this system. Although this would most likely not prove too difficult, the time it would require was better spent this year on improving the core artificial intelligence rather than creating new brake physics for a variety of vehicle models. Another possible improvement would be the implementation of deeper learning to find a method that was not thought of while creating this intersection. Deeper learning could possibly find a way to navigate the intersection that is much more efficient than the current method. Overall, however, this project was quite successful.

Acknowledgements

We would like to thank a friend of ours for suggesting the use of XNA and MonoGame to create my simulation, our computer teacher for advice on data transfer protocols, and our friends and family for their consistent support.

Works Cited

"About." *MonoGame*, MonoGame Team, www.monogame.net/about/.

"C# Collections." *Www.tutorialspoint.com*, Tutorials Point, www.tutorialspoint.com/csharp/csharp_collections.htm.

Dorigo, Marco, and Mauro Birattari. "Swarm Intelligence." *Scholarpedia*, Scholarpedia, 14 Jan. 2014, www.scholarpedia.org/article/Swarm_intelligence.

"Drag Coefficient." Engineering Toolbox, www.engineeringtoolbox.com/drag-coefficient-d_627.html.

"Lesson: Object-Oriented Programming Concepts." *The Java™ Tutorials*, Oracle, 2017, https://docs.oracle.com/javase/tutorial/java/concepts/.

L.W.C., Nirosh. "Introduction to Object Oriented Programming Concepts (OOP) and More. CodeProject, 4 Feb. 2015, www.codeproject.com/Articles/22769/Introduction-to-Object-Oriented-Programming-Concep.

XNA Frequently Asked Questions." Internet Archive: Wayback Machine, Microsoft, 28 Aug. 2008, web.archive.org/web/20090908145646/http://msdn.microsoft.com/en-us/xna/aa937793.aspx.

Miller, Peter. “Swarm Theory.” *National Geographic Magazine*, National Geographic Society, 5 Aug. 2007, http://ngm.nationalgeographic.com/2007/07/swarms/miller-text.

Mkhitaryan, Armina. “Why Is C# Among The Most Popular Programming Languages in The World?” *Medium*, SoloLearn, 13 Oct. 2017, https://medium.com/sololearn/why-is-c-among-the-most-popular-programming-languages-in-the-world-ccf26824ffcb.

“Rolling Resistance.” Engineering Toolbox, 2008, www.engineeringtoolbox.com/rolling-friction-resistance-d_1303.html.

Sherman, Don. “Drag Queens.” Tesla, pp. 86–92., www.tesla.com/sites/default/files/blog_attachments/the-slipperiest-car-on-the-road.pdf.

Stanbrough, J. L. “An Unbanked Turn.” BHS Physics, 4 Jan. 2005, www.batesville.k12.in.us/physics/PhyNet/Mechanics/Circular%20Motion/an_unbanked_turn.htm.

“Static Friction Formula.” Soft Schools , www.softschools.com/formulas/physics/static_friction_formula/30/.

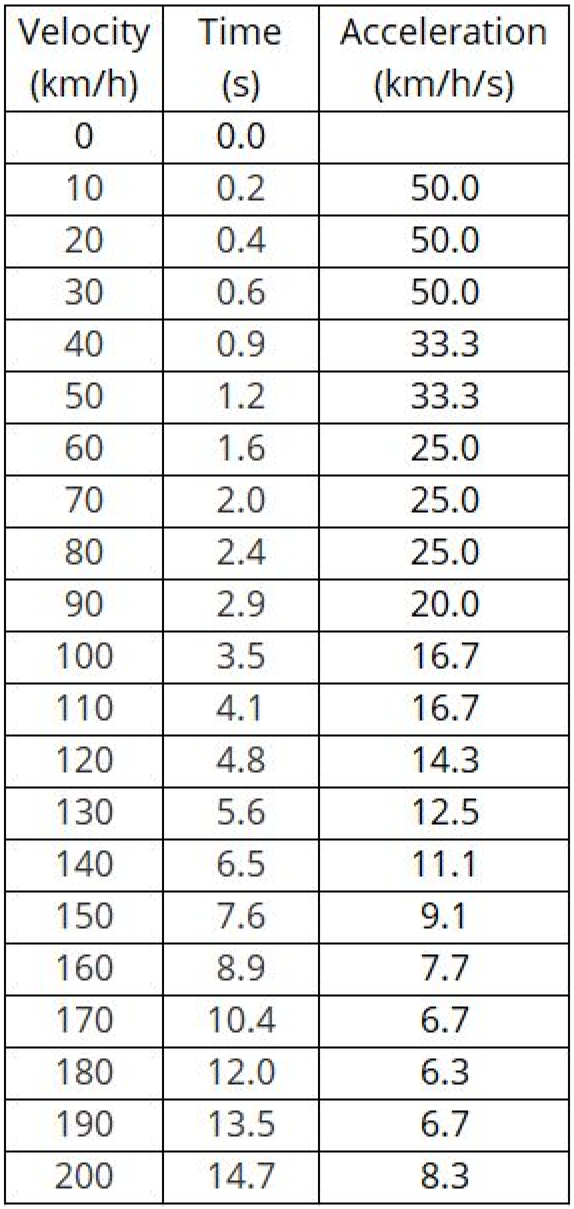
Stone, Peter, et al. “AIM: Autonomous Intersection Management.” *Project Home Page*, Kurt Dresner, 4 Dec. 2017, www.cs.utexas.edu/~aim/.

Wagner, Bill, et al. “Introduction to the C# Language and the .NET Framework.” *Microsoft Docs*, Microsoft, 20 July 2015, https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.

Whitaker, R B. “MonoGame Tutorials.” RB Whitaker's Wiki, Wikidot.com, rbwhitaker.wikidot.com/monogame-tutorials.

Appendix

Real-World Speed Data of Tesla Model S:



Velocity (km/h)	Time (s)	Acceleration (km/h/s)
0	0.0	
10	0.2	50.0
20	0.4	50.0
30	0.6	50.0
40	0.9	33.3
50	1.2	33.3
60	1.6	25.0
70	2.0	25.0
80	2.4	25.0
90	2.9	20.0
100	3.5	16.7
110	4.1	16.7
120	4.8	14.3
130	5.6	12.5
140	6.5	11.1
150	7.6	9.1
160	8.9	7.7
170	10.4	6.7
180	12.0	6.3
190	13.5	6.7
200	14.7	8.3

One Lane Intersection

1. Vehicles Turning Right:
    a. Vehicles From Left
        i. Straight
    b. Vehicles From Opposite
        i. Left
    c. Vehicles From Right
        i. None
2. Vehicles Going Straight
    a. Vehicles From Left
        i. Straight
        ii. Left
    b. Vehicles From Opposite
        i. Left
    c. Vehicles From Right
        i. Right
        ii. Straight
        iii. Left
3. Vehicles Turning Left
    a. Vehicles From Left
        i. Straight
        ii. Left
    b. Vehicles From Opposite
        i. Right
        ii. Straight
    c. Vehicles From Right
        i. Straight
        ii. Left

Multi Lane Intersection

1. Vehicles Turning Right
    a. Vehicles From Left
        i. In Right Lane
            1. Straight
        ii. In Middle Lanes
            1. None
        iii. In Left Lane (Closest to center line)
            1. None
    b. Vehicles From Opposite
        i. None
    c. Vehicles From Right
        i. None
2. Vehicles Going Straight
    a. Vehicle in Left Lane
        i. Vehicles From Left
            1. Straight
            2. Left
        ii. Vehicles From Opposite
            1. Left
        iii. Vehicles From Right
            1. Straight
            2. Left
    b. Vehicle in Middle Lane
        i. Vehicle in Middle Lane
            1. Vehicles From Left
                a. Straight
            2. Vehicles From Opposite
                a. Left
            3. Vehicles From Right
                a. Straight
                b. Left
    c. Vehicle in Right Lane
        i. Vehicles From Left
            1. Straight
        ii. Vehicles From Opposite
            1. Left
        iii. Vehicles From Right
            1. Right

            2. Straight
            3. Left
3. Vehicles Turning Left
    a. Vehicles From Left
        i. Straight
        ii. Left
    b. Vehicles From Opposite
        i. Straight
        ii. Left
    c. Vehicles From Right
        i. Straight
        ii. Left

The Code: