Design Studio 2 - Part 1

Thursday, Nov. 9, 2017

INF 121

Designers

Daniel Tryon: 20621204
Andre Ace Aquiler: 80163113
Daniel Lee: 48789422
Bryson Cateil: 83626339
Aishwarya Bhasin: 57802539
Omar Chaudhry: 39341547

Table of Contents

Traffic Signal Simulator: An Introduction

This application has been designed in order to aid students in understanding traffic signal timing as part of a course in civil engineering at the University of California, Irvine. Users will have the ability to create a map of connected intersections, set the timing of each one, adjust traffic density, and start the simulation. Once the user has started their simulation, they will see the results of their timing schemes in real time. The goal is for the user to find the optimal settings for smooth traffic while showing what can happen when their timing schemes are subpar.

A traffic simulation begins with a user defining the basic measurements of a square grid. This grid will be the map on which a user can place intersections in horizontal and vertical positions. Once they have placed these positions on the grid, they must click on a button to link these intersections together via roads. Because we have a constraint which does not allow for curving roads, the simulator program will align all intersections into appropriate horizontal or vertical positions if the user has not placed them as such. Once the grid has been populated with intersections and aligned to follow our constraint, the user will then be able to manipulate the various properties of the map.

The properties of our traffic signal lights can be changed to various degrees to fit our use specifications. All lights have timers that control for how long a red, yellow and green light stays lit before cycling to another state. There are left only turn lights which must also cycle between states, and there are right turn lights which follow this same state cycling system. These states will be available to manipulate by the user once the grid and intersections have been placed and correctly connect to each other. Red and yellow timers have base limits which users cannot change. These were placed to satisfy our constraint which does not allow for state combinations which would cause accidents. Users may choose to place sensors on intersections, which allows intersections to change their state durations based on whether there are cars still in lane.

Another key property of our simulation is user defined densities for car flows. User have the ability to control how many cars flow into the simulation at each entry point on the boundaries of the map. These densities will allow users to see how cars flow through the system given the timers they have chosen and the densities they have chosen. The user will have visual representations that show areas of extreme car densities, signifying traffic congestion, and areas of low car density, signifying low or no traffic congestion.

State Change Algorithm

The basic design of our algorithm begins by having request pulls from one end of an intersection to the other while a green light is on. If a car can fit in a lane it will move forward and enter the lane. This process will continue for every clock tick until the intersections green light timer has ended. For every state in which cars can move, those being forward movement, left turns, and right turns, timers set by the user will continuously allow for cars to progress forward

towards their destinations. Every clock tick of the simulation is global, and thus all intersections are updated to allow for movement in their lane before they send pull requests to other lanes. A more detailed discussion of the algorithm is listed under the algorithm section of our design.

The Educational Aspects of our design

Ultimately, users will be given information about how the various properties they chose during their simulation have affected the flow of traffic in the simulation. Visually, they will see areas of greater car density, indicating poor timers, and areas of lower car density indicating better timer configurations. All cars will have individual timers counting how long they have been stationary, and how long it took them to get to their destination. Intersections will display how many pull requests could not be completed, and how long cars spent idle in their lane. All this data will be presented to the user during and after the simulation is stopped, and users can change timers and states whenever they have the simulation stopped in order to try and better the efficiency of the simulation.

Audience:

- Civil Engineering Students

Stakeholders:

- Professor E.

Goals:

- Students must be able to control each signal's timing scheme
- Students must be able to create a visual map of an area, laying out roads in a pattern of their choosing
- Students must be able to add a sensor to any intersection; this will alter the signal timing based on where cars are stopped
- Traffic levels should be conveyed to the user in real-time
- The current state of the traffic lights should be shown (updated when changed)
- Students must be able to control incoming traffic density at each individual boundary road

Constraints:

- Combinations of traffic light states that would result in a car crash are not allowed
- All intersections must be 4-way and have traffic lights
- Users must set attributes for each Intersection (Boundary or Connecting) before the simulation can begin
- Only one simulation can run at a time
- All cars advance at the same speed
- Cars cannot pass one another
- All roads must be horizontal or vertical (no diagonal nor winding roads)
- Limit map width and height to the width of the display (desktop or mobile)

Assumptions:

- Users will have basic understanding of traffic lights (e.g. red = stop, green = go)
- Users' computers will be current enough to support our simulation
- Users are coherent enough to follow simple prompts (e.g. set locations, start simulation)

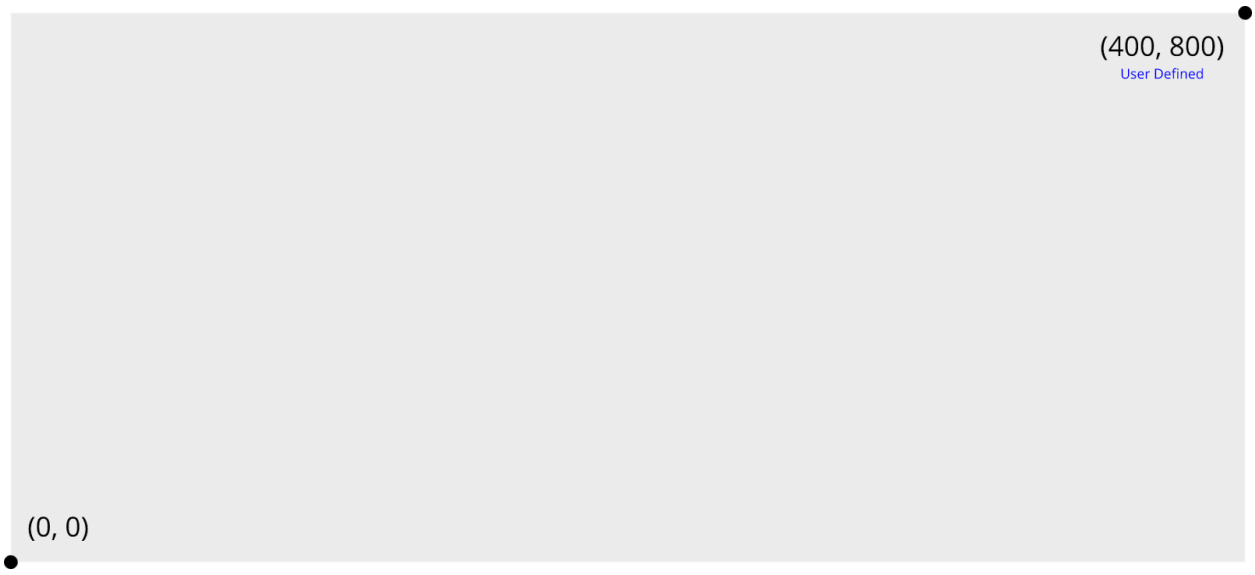
Decisions:

- The app doesn't stop by itself, the user must stop it manually
- No U-turns: requires unprotected right turns (autonomous cars)
- Timer will merely influence timer durations (simple)
- Yellow light has minimum of 3 cycles to account for breaking time
- All Red has minimum of 2 cycles to clear theoretical intersections
- Intersection does not take up physical space
    - If car crosses from one lane to the next, then it has passed the intersection

Walkthrough:

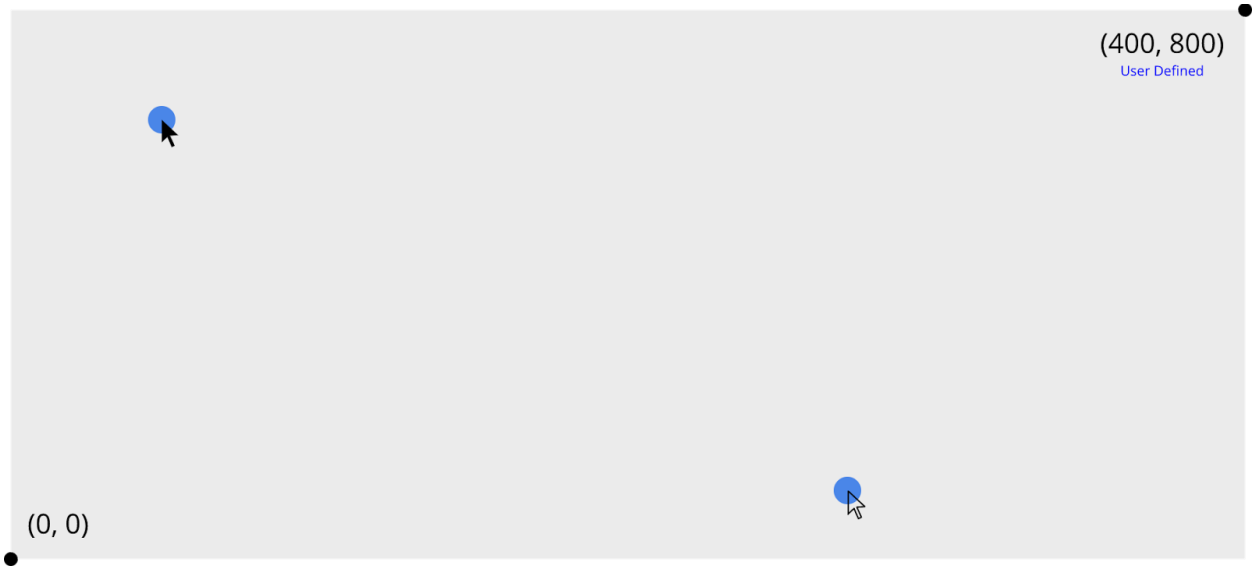
User Specifies Map Size:

At this stage, the user will be asked to enter the width and height of the map. These values are limited by the width of the display (each block is 10 pixels).



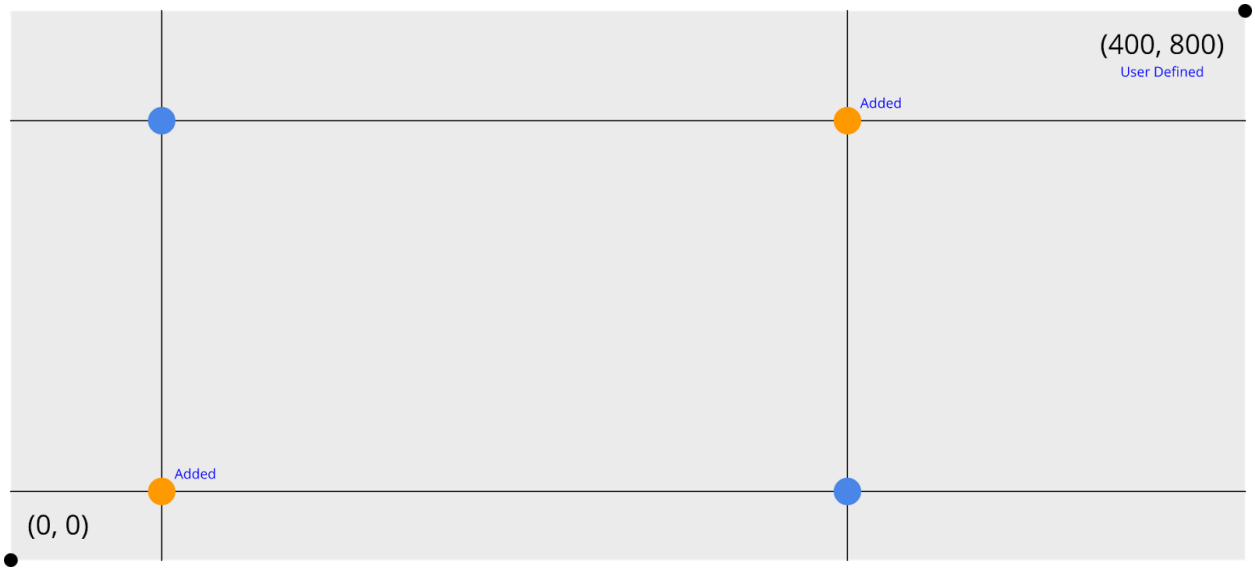
User Selects Intersection Locations:

Once the map size is set, the user can click anywhere on the map to place an intersection. The user can place as many intersections as will fit on the map.



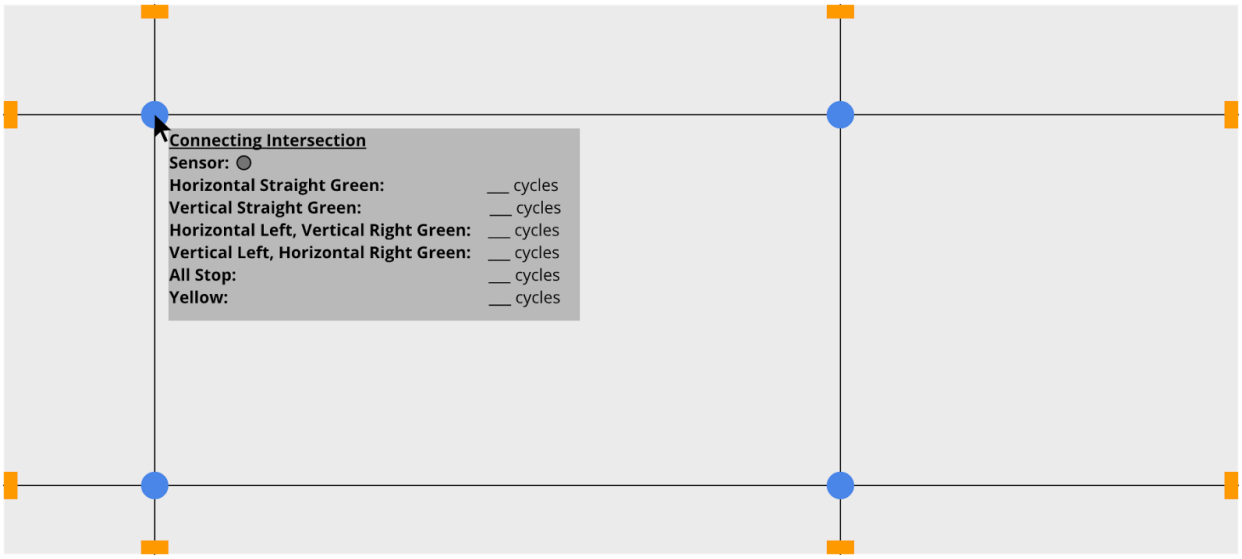
Map Connects Intersections:

Once the user has finished adding intersections, the application will add in intersections where roads cross.



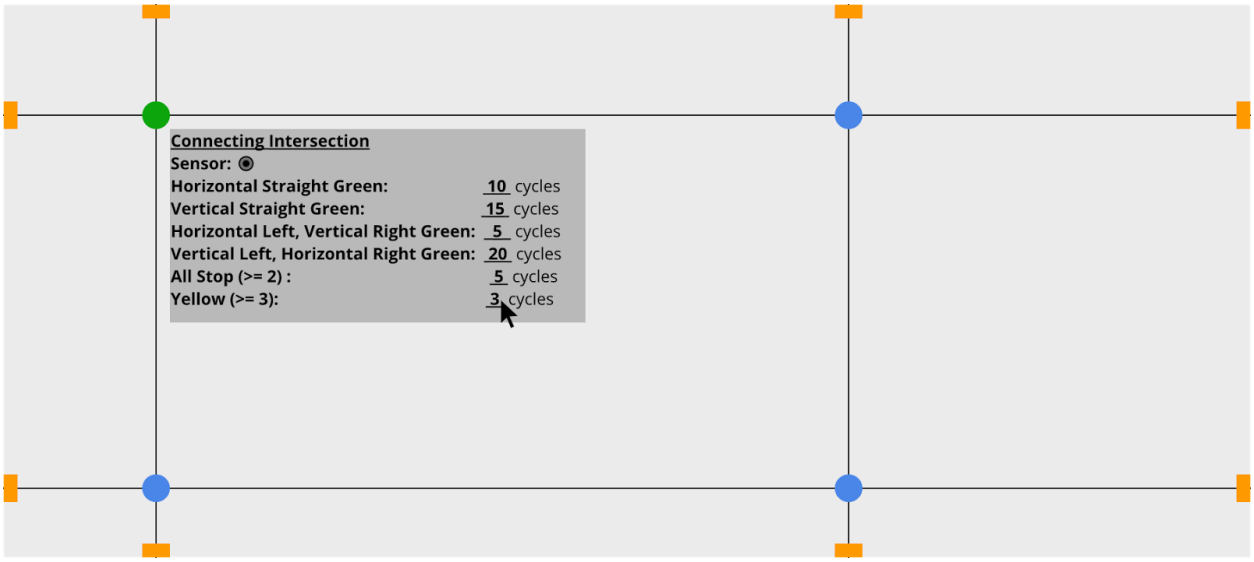
User Sets Connecting Intersection Attributes:

After the intersections are added by the application, boundaries are placed along the edges. The user is then required to set the attributes for each intersection.



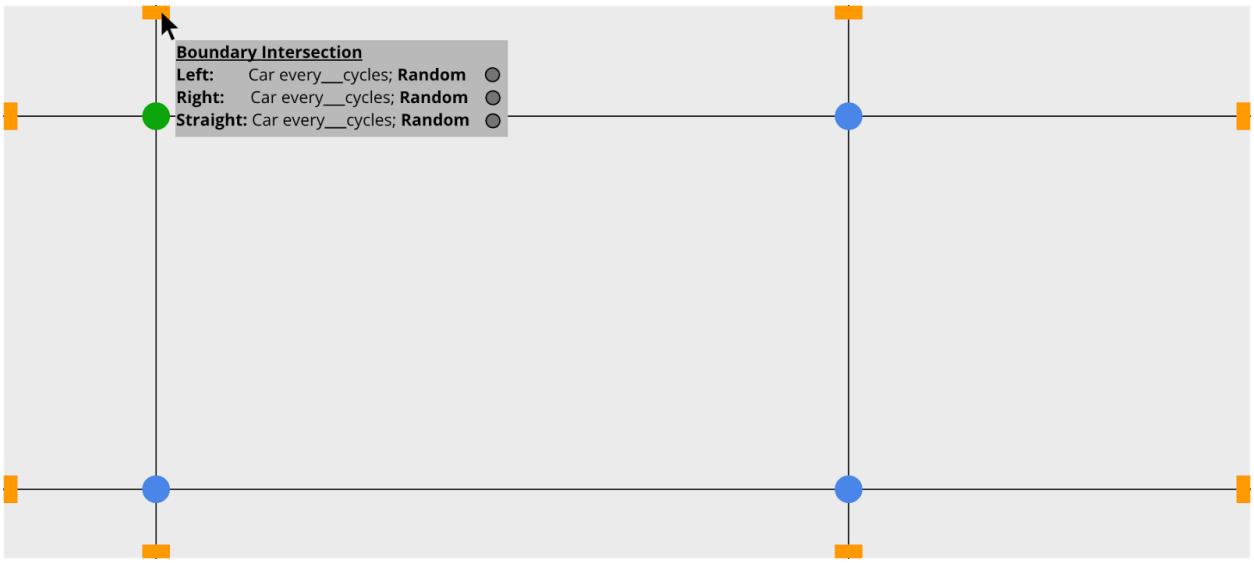
After Connecting Intersection Attributes are Set:

Here the user can decide whether or not to add a sensor, and how many cycles each status will last. Users will also be asked what the initial status will be (not shown). Once the required attributes for an intersection have been filled, the intersection turns green.



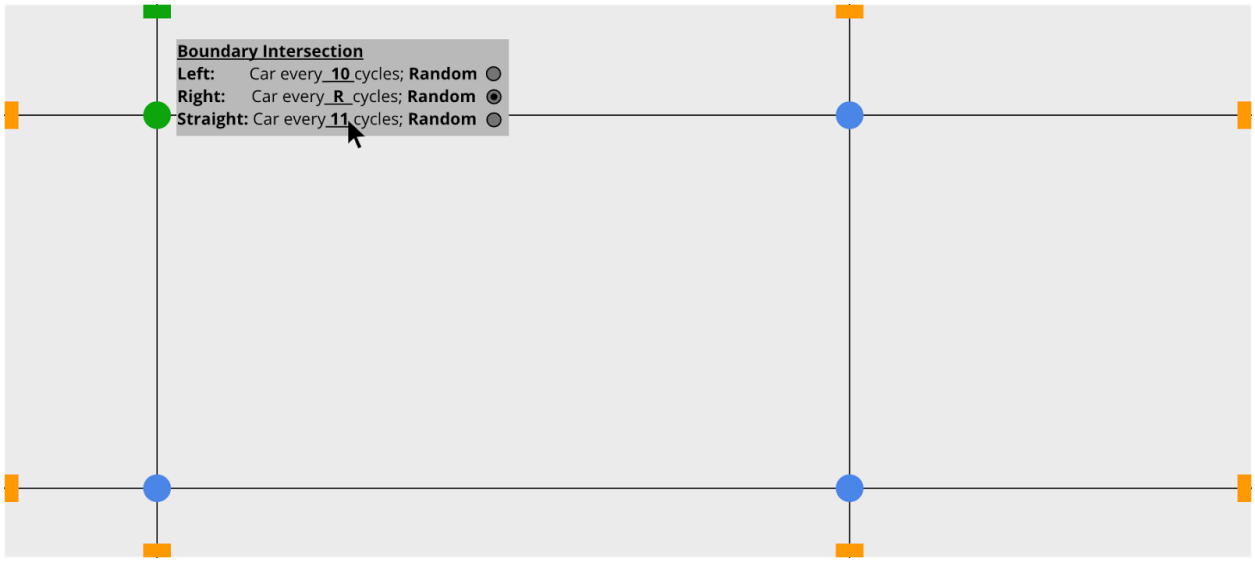
User Sets Boundary Intersection Attributes:

The user is also required to set all attributes for each boundary. Here the user can set the traffic density for each cycle. The value here will determine how many cycles must pass before a car is added in each lane. The random toggle, when turned on, will negate any value previously added by the user; the random function will replace the value every time a car is added (random cycles per car after every add).



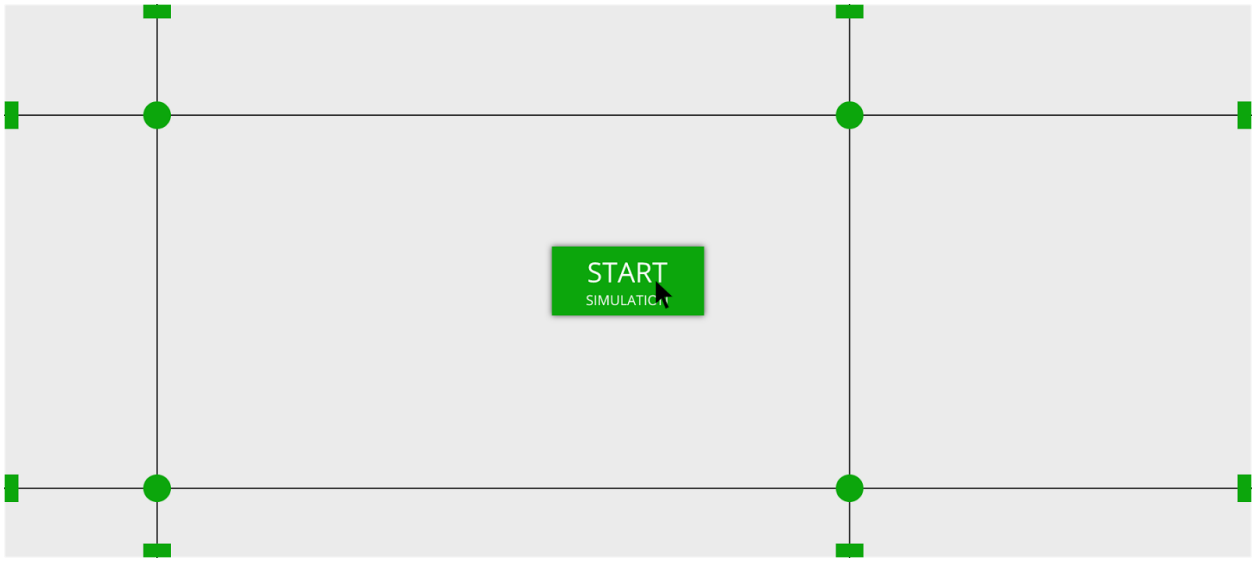
After Boundary Intersection Attributes are Set:

After all of the attributes have been set for a boundary, it will turn green. This signifies that no further action is required at this particular boundary.



After All Boundaries and Connecting Intersections are Set:

Once all boundaries and connecting intersections have been set, all nodes will appear green and a start button will appear. The start button, when clicked, will set the simulation in motion.

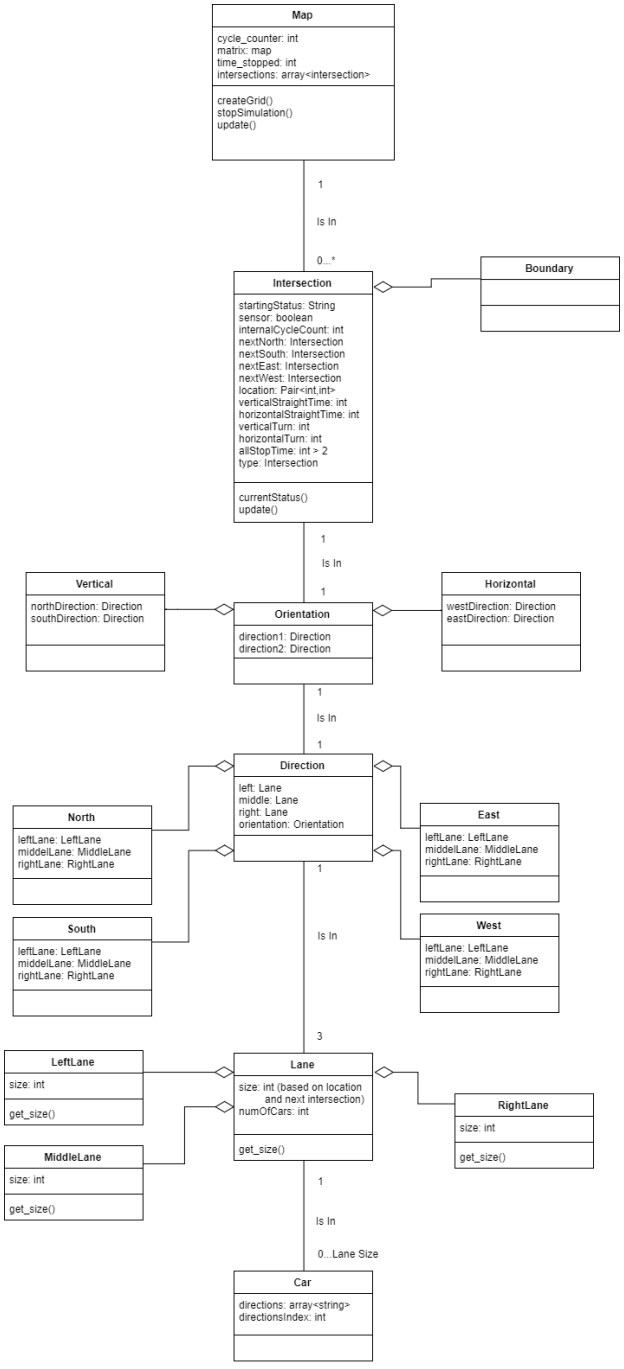


After Simulation Starts:

The users will be able to see the status of each light, along with the location of the cars. The cars will be displayed individually. The user has to manually stop the simulation in order for it to halt. After halting, the application will display the amount of cycles that all of the cars spent waiting at each intersection. The goal is to find map settings that will minimize these numbers. At this stage, the user will be able to change settings or start with a new map altogether.

UML Diagram:

The Unified Modeling Language diagram provided in this document displays the relationship between the data structures used in the application. Essentially, groups of information are stored in container objects which are then stored in other bigger, broader container objects. Many of the objects have base classes in which all iterations of will inherit attributes and methods.


Map
cycle_counter: int
matrix: map
time_stopped: int
intersections: array<intersection>
createGrid()
stopSimulation()
update()
1
Is In
0...*
Intersection
startingStatus: String
sensor: boolean
internalCycleCount: int
nextNorth: Intersection
nextSouth: Intersection
nextEast: Intersection
nextWest: Intersection
location: Pair<int,int>
verticalStraightTime: int
horizontalStraightTime: int
verticalTurn: int
horizontalTurn: int
allStopTime: int > 2
type: Intersection
currentStatus()
update()
Boundary
1
Is In
1
Vertical
northDirection: Direction
southDirection: Direction
Orientation
direction1: Direction
direction2: Direction
Horizontal
westDirection: Direction
eastDirection: Direction
1
Is In
1
Direction
left: Lane
middle: Lane
right: Lane
orientation: Orientation
North
leftLane: LeftLane
middleLane: MiddleLane
rightLane: RightLane
East
leftLane: LeftLane
middleLane: MiddleLane
rightLane: RightLane
1
Is In
South
leftLane: LeftLane
middleLane: MiddleLane
rightLane: RightLane
West
leftLane: LeftLane
middleLane: MiddleLane
rightLane: RightLane
3
LeftLane
size: int
get_size()
Lane
size: int (based on location and next intersection)
numOfCars: int
get_size()
RightLane
size: int
get_size()
MiddleLane
size: int
get_size()
1
Is In
0...Lane Size
Car
directions: array<string>
directionsIndex: int

Data Structures:

We implemented a map class, which represents different intersections that the user selects from the grid. The coordinates of the intersections represent locations in the map that are contained in a two dimensional matrix. The map contains a cycle counter that is incremented each time the map is updated. Each intersection specifies a starting status that tells whether the horizontal or vertical traffic will simulate first. Moreover, intersection calls the orientation class which takes in the starting status as its argument to determine if it is north/south or east/west. It also takes into account if the sensors are being used or not through a boolean expression. The orientation class consists of directions that has three lanes: left, middle, and right. Each lane has a size associated with it that shows the number of cars that it can hold. Also, the lanes are connected to each other through pointers in linked lists, just like the intersections. Lastly, each car on the lane takes up a space in the queue and each car can either go straight, left or right.

```
import Queue
Import random

class ListNode:

   def __init__(self, init_val, next_node = None):
      self.value = init_value
      self.next = next_node

class Car:

   def __init__(self):

      self.directions = ["right", "straight", "left"]


class Lane:

   def __init__(self, size, next_lane = None):

      self.size = size

      self.lane = customQueue(self.size)

      if len(self.lane) > size:

         raise LaneSizeError
```

```
        self.next = ListNode(self, next_lane)


class Direction:

    def __init__(self, orientation, left_size, right_size, straight_size
                , next_left = None, next_right = None, next_straight = None):

        self.orientation = orientation #if vert, 1 north 1 south, else 1 east 1 west

        self.left = Lane(left_size, next_left)
        self.right = Lane(right_size, next_right)
        self.straight = Lane(straight_size, next_straight)

class Orientation:

    def __init__(self, start_status, left_size, right_size, straight_size):

        if start_status == "Vertical":
            self.orientation1 = "North"
            self.orientation2 = "South"
        else:
            self.orientation1 = "East"
            self.orientation2 = "West"

        self.direction_1 = Direction(orientation1, left_size, right_size, straight_size)
        self.direction_2 = Direction(orientation2, left_size, right_size, straight_size)


class Intersection:

    def __init__(self, starting_status, x, y):

        self.x = x
        self.y = y
        self.cycle_count = 0
        if starting_status = "V":
            self.starting_status = "Vertical"
            self.opposite = "Horizontal"
        else:
            self.starting_status = "Horizontal"
            self.opposite = "Vertical"
```

```
        self.sensor = False
        self.traffic_counter = 0

        self.starting_direction = Orientation(starting_status, left_size, right_size,
straight_size)
        self.waiting_direction = Orientation(opposite, left_size, right_size, straight_size)


class Map:

    def __init__(self, intersections):

        self.cycle = 0

        self.time_stopped = 0

        self.intersections = intersections

        self.matrix = [[]]

        #insert map building code here
```

Algorithm:

After the user has started the simulation, the Map.update() function is called continuously until the student stops the simulation. Each time the Map is updated, it will increment the cycle counter and update all of its boundaries.

```
Map.update() {
        cycle_counter++;
        (Boundary n : north_boundaries) { n.update_north(); }
        (Boundary s : south_boundaries) { s.update_south(); }
        (Boundary e : east_boundaries) { e.update_east(); }
        (Boundary w : west_boundaries) { w.update_west(); }
}
```

The following update algorithm acts on all of the boundaries within the Map. North Boundary update will behave differently than South Boundary update (and East, and West for that matter). Using a North Boundary as an example, the cars in it's Northbound Lanes will be advanced. Iterating through the lane array, each car will move forward if there is no car in front of it. If a car cannot be moved, then the containing Intersection's attribute, time waiting, will be incremented. After all Lanes have been advanced, the Intersection will attempt to pull a car from its Next Intersection. If the next Intersection has cars turning left into the current Intersection, then a car will be pulled from the next Intersection's Westbound lane. If the next Intersection has cars turning right into the current Intersection, then a car will be pulled from the next Intersection's Eastbound lane. Lastly, if the next Intersection has cars going straight into it, then a car will be pulled from the next Intersection's Northbound lane. All of the states are mutually exclusive (e.g if a car is turning right into the lane, then another car cannot be coming from another direction).

After the attempt at pulling a car from the next Intersection, the current Intersection will become its next Intersection. This cycle will continue until the current Intersection's next is NULL. This signifies that the current intersection is a boundary. As soon as this happens, the current Intersection will still advance the cars in the usual way. However when it comes time to update, the lane will generate cars in each lane at the rate specified by the user (during Map setup) at the boundary.

Note: Boundary and Connecting are **types of Intersections**
north_boundary[0].next = the connecting boundary to the south
Each north_boundary.update() ends when there are no more connecting boundaries

```
north_boundary[0].update() {
        If this.type == Boundary
                Remove car from first spot

        For every Car in every Lane
```

```
                    If there is an empty spot in front of Car
                        Move Car forward
                    Else
                        Increment Intersection's time_waiting counter
            If there is a Sensor
                    If there is a car in the first spot
                        If the current lane is red
                            If the current_status != All_Red
                                Map.cycle_counter++

            While (Intersection.Next is not NULL) // south_intersection in this case
                    If Next.status is (Vertical_Left, Horizontal_Right),
                        A car will be pulled from the Westbound.Right Lane if there is a car in the first
                        spot

                    Else If Next.status is (Vertical_Right, Horizontal_Left)
                        A car will be pulled from the Eastbound.Left Lane if there is a car in the first spot

                    Else If Next.status is (Vertical_Straight, Horizontal_Stop)
                        A car will be pulled from the Northbound.Straight Lane if there is a car in the
                        first spot

                    Else If Next.status is (Vertical_Stop, Horizontal_Straight) OR (Vertical_Stop,
                    Horizontal_Stop)
                        No cars will be pulled.

                    Intersection = Intersection.Next

                    If car can be pulled: check car's next direction, and check if it's next lane is full.
                        If the spot is available, transfer the car and increment car.direction_index
                        Car.direction.index++
                        Note: If car has no more directions, car will go straight for the remainder of the
                        simulation

            If Intersection.Next is NULL, then this.type == Boundary // always true at this point
                    Add a car at the rate specified by the user at that Boundary intersection
    }
```

This is a top-down approach in which the Map's static counter trickles down into its Intersections. Each intersection has a current status, along with a list of next behaviors.

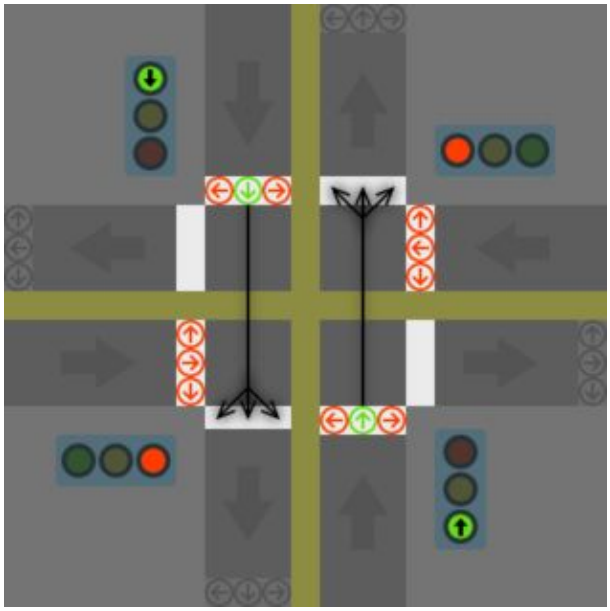
Status Cycle:

1. Vertical Left, Horizontal Straight
2. All Stop
3. Vertical Straight
4. All Stop
5. Horizontal Left, Vertical Straight
6. All Stop

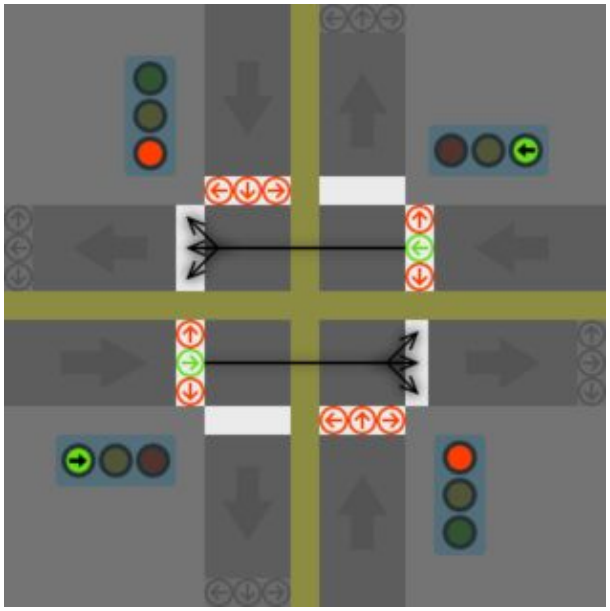
7. Horizontal Straight
8. All Stop
9. Repeat from 1

Each of these states has a user set timer. The time, derived from the Map's static counter, will dictate when the Intersection should change states.

Vertical Straight:

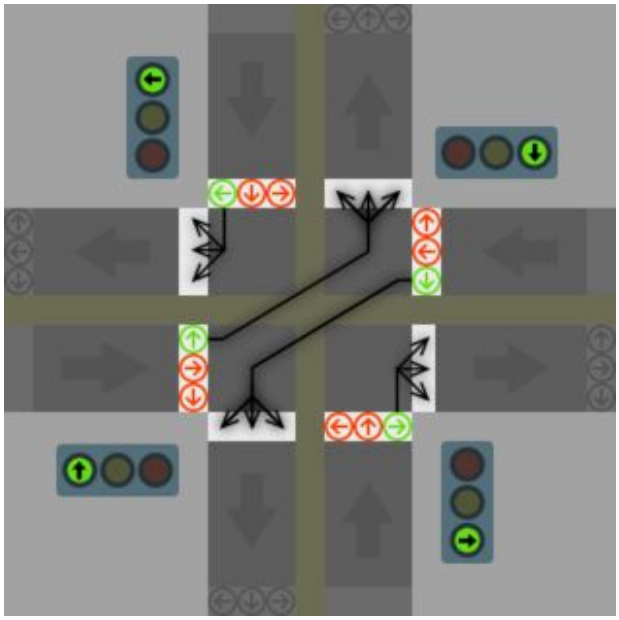


Horizontal Straight:

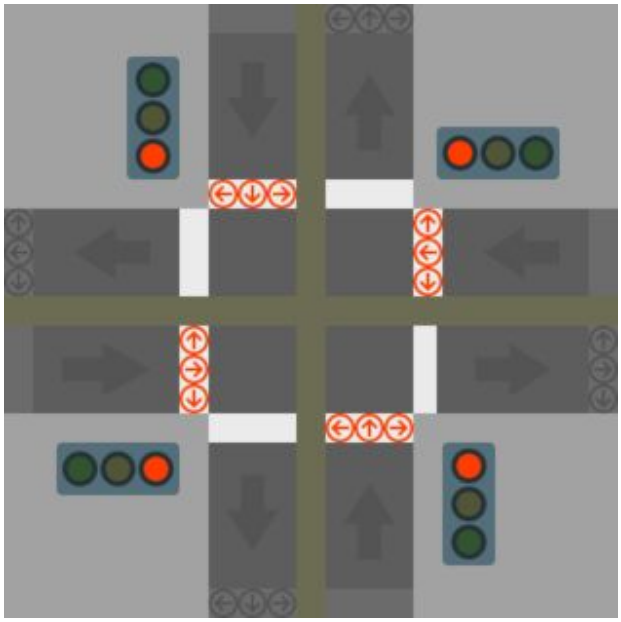


Horizontal Right, Vertical Left:

Vertical Right, Horizontal Left:



All Stop:



Alternative Approaches and Decisions:

Because of the scale of the project, we considered three different approaches for the final design. In deciding on the final functions and features for the software, we kept in mind the overall goals of the project. We agreed to make trade-offs based on our initial list of goals, and eventually chose the design we felt best addressed the needs of the end user, discarding the remaining two approaches.

Complex Approach

Differing Goals:

- A fully realistic representation of traffic via actual intersections
- Simple AI for individual cars to simulate actual driving conditions

Differing Constraints:

- Complexity of Simulation requires a computer with higher performance requirements
- May require more use of outside API's/other libraries we do not currently have access to
- Cars must navigate middle of intersection as well as lanes
- Cars can navigate separate lanes of an intersection as they please

Differing Assumptions:

- User has access to higher end computer.
- Students will benefit from having more accurate information at the expense of faster loading times

Differing Decisions:

- Remove intersections as possible places where cars could stop in. Removed them as data structures entirely.
- Removed fully autonomous cars, favouring a semi-random car destination approach

Differing Trade-offs

- Creating data structures representing intersections allows us to fully simulate gridlock as it is found in actual traffic conditions. These structures however, are much more difficult to implement due to the variation between states that they can be in.
- Cars which are fully autonomous simulate real world conditions, but are difficult to implement correctly. These cars take much more processing power, but more accurately represent traffic conditions.
- Allowing for more variables in the simulation creates a more realistic traffic simulation but some of the educational value of the simulation is lost. The user is expected to have a high degree of control over the timers of the intersections to influence traffic and learn from their decisions. By adding more variables outside of the user's' control we reduce the impact that they have on the system as a whole.

The first alternative design was an over-the-top approach which was heavy in detail and complexity. One key difference of this approach included an actual shared space for cars to exist in the middle of each intersection. Because each intersection is connected to by multiple lanes, which we designed as queues, it would create some over complication trying to format the data structures to all link to this intersection area. Instead, in our final model, we removed the area within each intersection, meaning that when a car reaches it, the car will immediately be sent to the bottom of its destination queue, representing the lane. Another feature we considered was a concept to give each car autonomy, or the freedom to make its own decisions, such as turning left, right, or making a U-turn when it was deemed to be safe to do so. In accordance with this concept, initially, we thought of creating a system without the use of left or right turn lights and the autonomy of the cars would allow for efficient traffic flow. However, we felt that having too high a level of detail could cause confusion, hindering the students' learning. As well as avoiding over complication, choosing a less elaborate approach would allow for the allocation of time and resources toward improving other sections of the software.

Simple Approach

Differing Goals:

- Students will be able to run simulations very quickly
- The simulation will return total time spent waiting for all cars (sum total)

Differing Constraints:

- Cars has no “time_waiting” value
- Cars have no “direction” array

Differing Assumptions:

- Students will benefit more from quick results rather than slow, more precise results

Differing Decisions:

- Cars stored as boolean values
- Cars hold no intrinsic information
    - Time spent waiting is a global variable
    - Direction is determined randomly by the Intersection during update

Trade-offs:

- Having cars represented by boolean values as opposed to having car objects would allow for the simulation to run faster and would require less space.
- The value returned to the user would be the global time spent waiting (not per intersection). This would also reduce time and space complexity as well as keep the interface simple. This gives less information at a quicker rate.
- This approach encourages the user to quickly perform simulations, one after the other, instead of carefully planning out each timing scheme.
- This will lead to more results overall, but they will be less precise.

We then considered an approach on the opposite end of the spectrum, a more simplistic design. Our second alternative focused on a more user-friendly experience, cutting out certain features to create a simpler, cleaner design. This system would provide the basic functionality required with a familiar and easy to use interface, while being efficient with resources. For example, we considered implementing cars in the software as a simple Boolean value within the lanes, eliminating any intrinsic data that they would have carried. Additionally, this simple approach would solely utilize a global timer for every traffic light on the map. Every intersection light would follow an integer value set by the user for its red and green timings. The problem with this path is that although it would make for a straightforward and intuitive design, it would go against one of the main goals of the project, that that traffic should flow as efficiently as possible. Obviously different intersections should have different light timings based on their traffic density in order to

make movement more fluid. Therefore, it became a priority to give the users the option to change the timers for each individual pair of traffic lights.

Our Approach

Trade-offs:

- Storing cars as objects rather than boolean values allowed us to store valuable information. Conveying this information to the user provided enough educational value to outweigh the extra time and space required.
- Cars will not have autonomy which means that there are no unprotected turns. Although this feature would have provided some extra information to the students, the educational value was not high enough to warrant the extra computing time required for this feature.
- The time spent waiting will be returned on a per intersection basis. Returning the value for each car would be too much information for the student to digest in any meaningful way. On the other hand, returning a global value would not allow the student to understand which intersections are performing better than others. Returning on a per intersection basis is the perfect blend of returning valuable and meaningful information to the user, while not requiring too much extra time and space.

Ultimately, we decided upon a balance between the two designs, weighing the pros and cons of each. We addressed what we felt were the most important features the software should contain and how it would satisfy the needs of our target audience without going overboard with design choices. In our simplistic design, we felt that while it concisely solved our problem, it did not provide enough feedback or return enough information to be educational to the students. And when we examined our more heavily detailed approach, we decided that it contained too much information and too many options that it may prove to be difficult to use, potentially turning the students off to the idea of using it.