William Yang
Dennis Li
Payam Dowlatyari
Kenzo Spaulding
Tyler Mos

Software Design Studio 2 Part 1

Designing a Traffic Simulation App

Table of Contents

App & UI Design:

Goals:

- Students must be able to create a grid-like visual map of an area, including at least six intersections, modifiable road lengths and intersection variance.
- Students must be able to specify and adjust the behavior of traffic lights.
- Students should have the option to install sensors on intersections.
- During the simulation, students can observe the simulation with visualization.
- Students should be able to change the incoming traffic density.
- The application should provide an opportunity for students to try different traffic scenarios to learn from the outcome of each one.
- The application must provide Civil Engineering students with their specific educational outcomes.
- Students must be able to observe any issue with their map's design, change it, and see the results of their correction.

Essence:

- The software must be designed to educate students clearly on how intersection light timing affects traffic flow.
- An introduction for students to learn about traffic science at the highest and most abstract level. It will visualize how traffic can ebb and flow from very small changes in light timings or sensor values, even if almost all of the real-world issues with traffic are removed from the situation.

Constraints:

- All roads are either horizontal or vertical - no diagonals or curves.
- Only 4-way intersections may exist - no T-shaped intersections or dead-ends - no one-way roads.
- Every intersection on the map must include traffic lights.
- Cars all go at the same speed across the entire simulation.
- Users must be presented with information relating to the traffic density (number of cars) of the roads.
- Only one simulation may be run at a time.
- No car crashes are allowed to occur.

- Some lights will have access to the cars that are queueing at the light, but not all of them.
- Students must be able to design each intersection with the option to have detection sensors or not.
- There are no pedestrians crossing the roads.

Assumptions:

- When a light turns green, cars will not accelerate to their expected speed. They will instead automatically move at the predetermined speed instantaneously.
- All cars advance at the same speed and they do not pass each other.
- Drivers follow the traffic rules and do not perform unusual behaviors.
- Cars will be the same size as one another.
- Cars will be the same weight as one another and all be able to trigger the sensor.
- Sensors sense the existence of cars in an intersection.
- The simulation acts under California traffic laws.
- UCI interpretation of traffic light (Cars cross when the light is green, cross if it is already in the intersection on orange, and stops at red)
- Cars drive on the right side of the street.

Ideas:

Application

- Allow cars to make turns
  - Turns can be accomplished via random number generators (with probability cut-offs generated by how busy a road is) or something similar to every third car or something like that
  - Turns can be implemented via random percentage to turn at any intersection for the cars or the intersection
- Don't allow cars to make turns
  - Non-realistic but easier to implement
  - Traffic science is really complicated so this really isn't a huge assumption when compared to the other assumptions made in this software
  - This can be accomplished by allowing a certain number of cars to go through the intersection at a certain rotation
- Make a time setter so the user can single-step through the traffic flow (crucial when traffic meets critical condition)

- Have split screen setup where one changes the states and such and another one that shows the compiled version running.
- The state of the simulation can only be set before the simulation is run, not while it is being run.
- Have compile-time errors occur if there are cars that will be running into each other
- Give the light access to the #of cars queued. (any cars queued or sensors are able to capture the number of cars?)
- Central clock
- Alternatives focus on the inclusion of turning.
- Able to wipe the scene when lights are changing and change the light configuration.
- Have preset features that can easily display to students what conditions are ideal or no-ideal
- Estimate feature that have a rough estimate of how much time will be added for each modification to the original state.
- Assuming that the professor will showcase the software in class, do we add status of roads? I.E add how “busy” a road is by calculating the time that a vehicle is on a certain road segment

Architecture

- Objects (OO architecture)
    - Map
        - Nested list of roads and empty space that allow for checking the interactions between roads and cars.
    - Cars
        - Traffic Level Affected by number of cars
    - Roads/Intersections
        - Roads are built in small car sized segments that are each filled or not
    - Sensors (pressure, determine placement from the light? , number of plates per road)
    - Traffic lights
        - Finite state machines for which cars are allowed to go, set on timers or sensor inputs which are set by user.
    - Current Time
        - Cars and lights act as a function how time has progressed
- Make it easy to add in turning at a later date

- User Inputs
    - How Lights Operate (Time Based Patterns, Pressure Sensors)
    - Quantity of Cars at a given road
    - When Cars Begin existing in the simulation
    - Length of Roads

    - Number/Placement of Roads and Intersections
    - Relative Time of Car speed to Traffic Lights in time based traffic light scenario
    - Percentage a car turns at any intersection (given the cars are allowed to turn)
    - Users should be able to calculate how fast lights shift between states.

Audience:

- Students: Students will use this program in a lab setting to see how traffic occurs in real-time.
- Professors and TAs: The professor and TA will likely walkthrough students how to operate this program.
- IT Staff: It is likely that tech staff will need to install this software onto computers, as it is unlikely that lab computers will come prepackaged with this software. They will also be responsible for required to update and maintain
- Front-end developer: As we are designing everything but interaction design, front-end developers will be interacting this program, and try to further design to match the needs of the academy.

Stakeholders:

- UCI: The university will be the entity who distributes the program amongst lab computers and those who plan to use it at home.
- Students: Students will use this program in a lab setting to see how traffic occurs in real-time.
- Professors and TAs: The professor and TA will likely walkthrough students how to operate this program.
- IT Staff: It is likely that tech staff will need to install this software onto computers, as it is unlikely that lab computers will come prepackaged with this software. They will also be responsible for required to update and maintain

Decisions:

- Central Clock ticks: The user will be able to increase/decrease the rate of the clock..
- Changing the current timing of the lights will wipe the scene (can have nice animation change effect)
- Split screen. One side shows how we set it up while the other side is the animation. Compiling one causes the animation to have all the cars run off and then the compiled one is painted and run.
- The implementation that only allows protected right turns has been selected.

- We choose to limit the amount of cars that can be queued at once to ensure the system does not back up.

Features Shared In All Forms:

We make the simulation through a map that is composed by road segments ordered in a grid-like manner. Each road segment can either be an intersection, where crossroads interaction and traffic light simulation takes place, a road end, where cars are spawned to drive to the other side of the road, or a normal road that can contain a vehicle. Students will be able to first create several road segments on an empty map, including intersections where they cross, then configure each intersection in terms of factors such as the frequency of car spawning, the timing of a traffic light, or the placement of a sensor. To prevent crash event from happening, the implementation have coded a set of restrictions that the user will be notified when configuring the intersection in the "wrong " way. The program also automatically adds road segments around an intersection so there can only be 4 way intersections.
The design of the architecture of making the simulation a grid-like map allows students to have an easy start by working in a framework, and will also make it easier for interaction developers to implement by not needing to design advanced arts, such as curved roads. The front-end designer can simply make art for every possible state of a road segment, then align it on the grid for a simulation.

Application Design Chosen : 2 (Right Turns)

Alternative 1: No turns

This design is a simple version of the software where cars will not be allowed to turn, and will drive straight once spawned. This design would be best at introducing students to the beginnings of traffic science. As traffic science is such a complex (and in some cases dense) area of civil engineering, a drastically simplified and controllable version would be beneficial to abstract away some of the more daunting sides of science.

Alternative 2: Right turns

Our second alternative design is a slightly more complex version of the first. In this alternative version the cars will have all of the features included in the first design, however there will also be the added functionality that will allow cars to make right turns. While adding turning does make the simulations a little more complicated, a car that is right turning does not operate too differently from one in which the car only goes straight. This implementation would allow cars to make right turns only on green lights. It would not require the addition of a new stoplight, as opposed to the addition of left turns would.

This is the Alternative we decided upon. This was because we found it the most comfortable space between complexity and abstraction. It would give students the ability to make a lot of choices for an intersection/road map pattern but also keep the situation simple enough to be easily understood and not overwhelming.

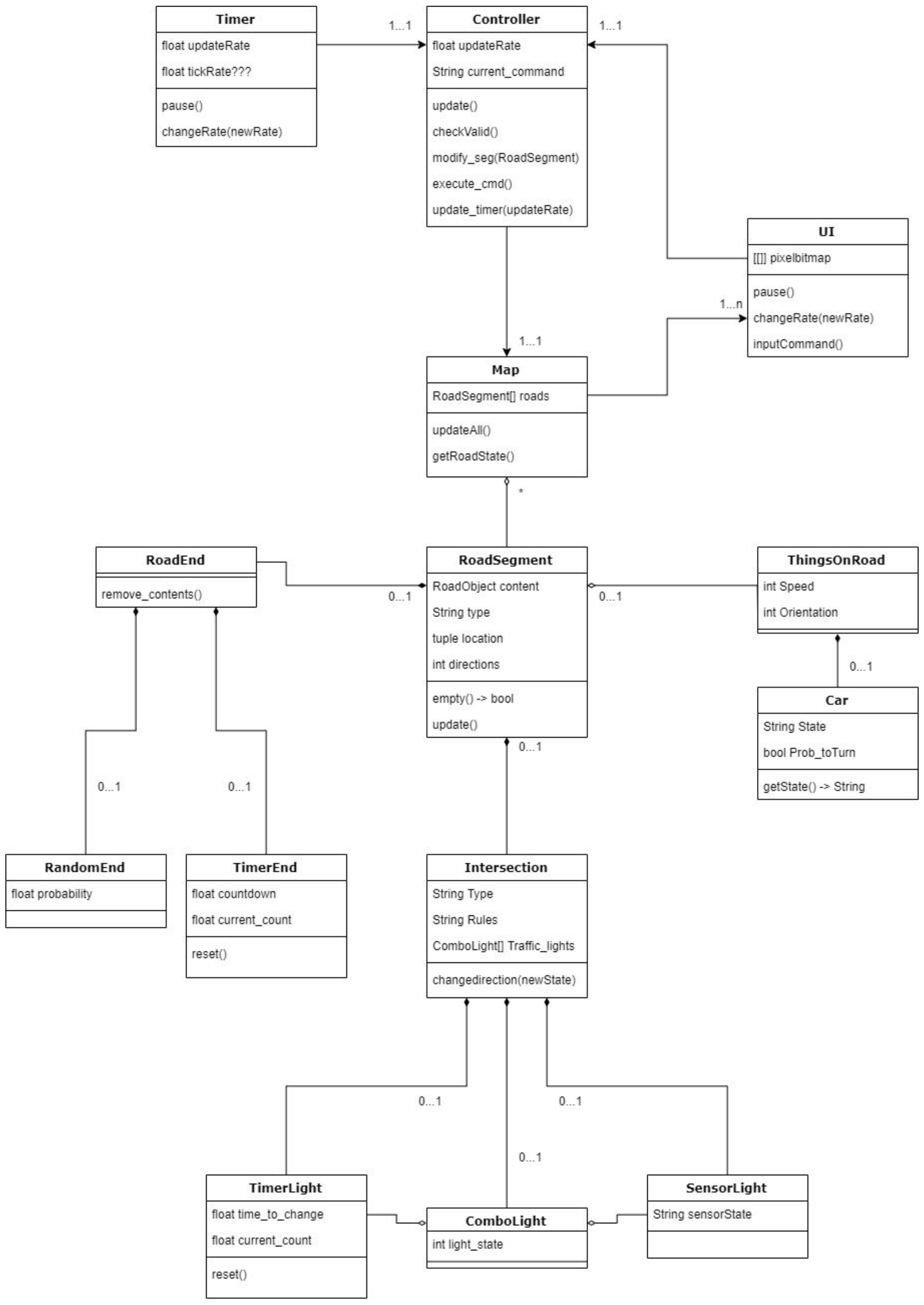
Alternative 3: Left and Right Turns

The most complex design alternative is one that includes the management of both left and right turns in our simulation. The main concerns when dealing with left turns in the simulation is managing both protected left turns, which requires an extra turn lane and extra logic to manage the extra state, as could be specified by the user. The second concern is figuring out how to deal with unprotected left turns. The logic and system design would inevitably become focused on designing a real time system where actors are able to predict the future locations of other cars/objects. This would result in large amounts of coupling and heavy computing for every road object in play.

While this alternative would be more realistic to real traffic, it would be likely detract from the real intent of the software; to be a simple and intuitive introduction to traffic science. With too much freedom for manipulation, students may be overwhelmed by the complexity of the intersections. Additionally, traffic science is incredible complex far beyond the extend of left turns or no left turns as concerns of pedestrians, cyclists, traffic accidents, road construction, variable traffic rates, and other common real life events drastically affect traffic. Thus imagining a scenario without left turns is assumption that will abstract away even more of the complexity to give a student a safe, and manageable entrance into the complex world of traffic science.

Architecture & Implementation Design

UML Diagram



Interface Elements:

Timer:

The Timer object will have the functionality of acting as a rate timer for which objects will flow through the simulation. The timer will be a variable tick rate that users will be allowed to modify. A faster rate will cause objects to move through the simulation faster, while a slower rate will do the opposite.

Timer.pause(): will pauser the current timer object. All of the objects that depend on the timers rate will stop moving.

Timer.changeRate(newRate) : will pass a newRate as a parameter and change the current rate at which objects are moving through the simulation.

Controller:

The controller will take in commands from the UI object, use them to update the current_command string and then output the current_command to the Map object.

Controller.update(): will allow the user to update the command currently stored in the controller.

Controller.checkValid(): will check if the currently stored command is a valid command.

Controller. modify_seg(RoadSegment): will pass a road segment modification to the map object.

Controller.execute_cmd(): will execute the current command stored in the Controller object.

Controller.update_timer(updateRate): will passing a new rate to be updated in the Map object.

UI:

The UI (user interface) will be a pixel bitmap that will display the positions of all of the objects associated with the Map() object.

UI.pause() will allow users to pause the current display of the map.

UI.changeRate() will allow users to change the tick rate at which all objects in the UI are progressing.

UI.inputCommand() will allow users to input commands to the map, such as adding more cars at an intersection or changing the timing of traffic lights.

Map Elements:

Map:

The Map object serves as the "backboard" of the interface. Only a single map will exist per simulation, and serves to hold the location of all the RoadSegments via coordinates, which in turn note the position of cars.

Map.updateAll() updates all the positions of the vehicles according to the expected algorithm below. This likely occurs each clock tick of the simulation.

RoadEnd:

The RoadEnd object is a parent object to RandomEnd and TimerEnd. All RoadEnd objects are, as the name implies, objects found at the end or road that spawns cars depending on what type of RoadEnd object it is.

RandomEnd:

The RandomEnd object spawns in cars at a rate determined by the user.

RandomEnd.probability indicates the probability a car is spawned.

RandomEnd.timeBetweenRolls indicates the time between each probability roll. This value can be optional, as the default can be set to a value like 1 tick.

TimerEnd:

The TimerEnd object spawns in cars as a function of time and ticks.

TimerEnd.countdown indicates the time remaining it takes for a car to spawn at this section of road.

TimerEnd.current_count indicates the remaining amount of cars left to spawn in.

The *TimerEnd.reset()* function resets the countdown and current_count to their original values.

RoadSegment:

RoadSegment is the parent class of both intersection and road-ends that is the basic element that builds up the simulation map. RoadSegments transfer ThingsOnRoad objects such as cars to neighboring RoadSegment objects to simulate traffic flow. Road Segment can include 2 car objects that goes in the opposite way of the same axis.

RoadSegment.content denotes what kind of object are on that particular RoadSegment object. In this specification this only includes the car object.

RoadSegment.type denotes the type of RoadSegment it is. For example, it could be a regular section of road or an intersection.

RoadSegment.location holds the expected coordinates of the RoadSegment object with respect to the Map.

RoadSegment.direction indicates which axis the car is aligned to. If aligned to the x-axis, cars travel horizontally. If aligned to the y-axis cars will travel vertically.

RoadSegment.empty() returns false if a RoadSegment currently has an object in the content parameter.

RoadSegment.update() moves the vehicle stored in the content object to the next neighboring empty RoadSegment, if able.

ThingsOnRoad:

The ThingsOnRoad object is a parent class to all objects that may appear on a road. In this current implementation, this object only includes the car object, but may expand into more diversely-sized vehicles like trucks, vans, etc. While this specification does not make use of non-moving objects, this parent object could be expanded to house those.

ThingsOnRoad.Orientation indicates the direction that a vehicle is traveling - along the X-axis or Y-axis.

ThingsOnRoad.Speed indicates the speed at which a vehicle is traveling. While vehicles in the current specification do not travel at varying speeds, this speed element can denote which direction a vehicle is traveling along an axis. For example, a negative speed denotes a vehicle traveling left along the X-axis or downwards along the Y-axis, while a positive speed denotes a vehicle traveling right along the X-axis or upwards along the Y-axis.

Car:

A car object is the most expected element to be used for this specification. Along with the orientation and the speed parameter it inherits from the ThingsOnRoad class, it also comes with one more parameter:

Car.state indicates whether a car has stopped moving or is currently moving along the road.

Intersection:

An intersection is a form of RoadSegment that serves as a place that both holds a vehicle and houses the stoplights. It can pass car objects in all four directions, and only occupies one space on the Map.

TimerLight: A form of intersection that uses stop lights that only act as a function of time.

ComboLight: A from of intersection that uses stop lights that act on time and accelerates slightly if a car is on a sensor.

SensorLight: A form of intersection that uses stop lights that act on whether or not a car is current on a sensor.

Pseudocode/Algorithms:

Algorithm for changing Map:

```
//Make the map a nested list of empty space, then use this function to add road_segments on //the map
If(the type to change is empty, so user want to delete a road segment)
    Map[x][y] = empty;
    //change nearby road segments into road ends
    For each adjacent block Block:
        If Block it is not empty: change Block's type to road_ends
If(the type to change is road segment)
    Map[x][y] = roadsegment with orientation (optional_orientation);
    Orientation_list = [];
    For each adjacent block Block:
        If Block's type is road_segment:
            Append Block's orientation in Orientation_list
    If variatety_of(orientation_list) == 2: // Map[x][y] is an intersection
        Map[x][y].type = intersection;
        //autogenerate adjacent grids as there can only be a 4 way intersection
        For each adjacent block Block:
            If Block's type is empty:
            Change_map_state(Block's x, Block's y, road_segment, orientation)
    If size_of(orientatinon_list) == 1: // Map[x][y] is a road_end
        Map[x][y].type = road_end;
```

The map is a nested list of roadsegment objects. This grid is the core of the architecture because everything either interacts with, or composes the map. It has some core functionality for verifying that the inputs made are valid and automates the composition of intersections and road ends based on the ends of a line of road segments or at intersections of two road segments of opposing directions. As road segments are objects that have two spaces for thingsonroad objects (one in each direction), intersections will also have two occupancy spaces and the allowed movement direction will be determined by the logic specified by the user.

Algorithm for Intersection Lights:

```
Light's State = X axis Green and Y axis Red;
Count = 0 time in ticks;
If (lights have been set to a time based pattern only){
    While (simulation is active){
        Count++;
        If (Light's State == X axis Green and Y axis Red && Count ==
    timeBetweenGreenToOrange){
        Light's State = X axis Orange and Y axis Red;
        Count = 0 time in ticks;
    }
        Else if (Light's State == X axis Orange and Y axis Red && Count ==
    timeBetweenOrangeToRed){
        Light's State = X axis Red and Y axis Green;
        Count = 0 time in ticks;
    }
```

```
            Else if (*Light's State* == **X axis Red and Y axis Green** && *Count* ==
        **timeBetweenGreenToOrange**){
            Light's State = **X axis Red and Y axis Orange;**
            *Count* = **0 time in ticks;**
        }
        Else if (*Light's State* == **X axis Red and Y axis Orange** && *Count* ==
        **timeBetweenOrangeToRed**){
            Light's State = **X axis Green and Y axis Red;**
            *Count* = **0 time in ticks;**
        }
        }
}
```

A timer is constantly active and advances with each tick. If a light is green, it changes to orange once a decided amount of time is met. If a light is orange, it changes to red once a different decided amount of time is met. A red light shifts to green once once the light on the other axis has moved from green to orange to red. The timer is then reset.

```
////////////////////////////////////////////////////////////////////////
Else if (lights have been set to a sensor pattern only){
        While (**simulation is active**){

            *//Increments*

            If (*ClosestSensorOnYAxisTriggerred  && light's state* == **X axis Green and Y axis
        Red**){
            Count++;
        }
            Else If (*ClosestSensorOnXAxisTriggerred  && light's state* == **X axis Red and Y axis
        Green**){
            Count++;
        }
            Else if (light's state == **X axis Orange and Y axis Red ||** light's state == **X axis Red
        and Y axis Orange**){
            Count++;
            }

            *//Changes*
            If (Count == timeBetweenOrangeandRed && light's state == **X axis Orange and Y
        axis Red**){
                Light's state = **X axis Red and Y axis Green;**
                Count = 0;
        }

        Else If (Count == timeBetweenOrangeandRed && light's state == X axis Red and Y
    axis Orange){
            Light's state = X axis Green  and Y axis Red;
            Count = 0;
        }

        Else If (Count == requiredSensorTime && light's state == X axis Green and Y axis
    Red){
            Light's state = X axis Orange  and Y axis Red;
            Count = 0;
        }


        Else If (Count == requiredSensorTime && light's state == X axis Red  and Y axis
    Green){
            Light's state = X axis Red  and Y axis Orange;
            Count = 0;
        }

}
```

A timer is constantly active per intersection. The timer advances when the light is orange or when a car remains on a sensor. Once the timer has advanced to a certain point decided by the user, the light changes and the timer is reset.

```
/////////////////////////////////////////////////////////////////////////
Else if (lights have been set to a sensor pattern as well as time){
    While (simulation is active){
        Count++;
        If (Light's State == X axis Red and Y axis Green &&
ClosestSensorOnXAxisTriggerred){
            Count++;
        }
        Else if (Light's State == X axis Green and Y axis Red &&
ClosestSensorOnYAxisTriggerred)
        Count++;
    }

    If (Light's State == X axis Green and Y axis Red && Count ==
timeBetweenGreenToOrange){
        Light's State = X axis Orange and Y axis Red;
```

```
                Count = 0 time in ticks;
        }
                Else if (Light's State == X axis Orange and Y axis Red && Count ==
        timeBetweenOrangeToRed){
                Light's State = X axis Red and Y axis Green;
                Count = 0 time in ticks;
        }
                Else if (Light's State == X axis Red and Y axis Green && Count ==
        timeBetweenGreenToOrange){
                Light's State = X axis Red and Y axis Orange;
                Count = 0 time in ticks;
        }
        Else if (Light's State == X axis Red and Y axis Orange && Count ==
        timeBetweenOrangeToRed){
                Light's State = X axis Green and Y axis Red;
                Count = 0 time in ticks;
        }

}
```

A timer is constantly active per intersection. The timer advances per clock tick. However, if a car remains on a sensor, the timer advances once more per tick a car remains on the sensor. The light will change once the timer is equal or over a pre decided time, resetting the timer.

Algorithm for Roads

```
If (!this.empty){
        If (eitherAdjacentSpaceEmpty(Orientation))
                This = RoadEnd(this);
//At roadend leaving simulation
        Else If (currentSpace.type == RoadEnd && nextAdjacentSpaceToMove.type == null){
                Remove car from simulation;
        }
//At road
        Else If (nextAdjacentSpaceToMove.empty() and nextAdjacentSpaceToMove.type !=
Intersection){
                moveCarToAdjacentSpace()
        }
//At Intersection
        Else If (currentSpace.type == Intersection && nextAdjacentSpaceToMove.empty() &&
!rightAdjacentSpaceToMove.empty()){
```

```
                This.contents = null;
                nextAdjacentSpaceToMove.contents = Car;
        }
        Else If (currentSpace.type == Intersection && !nextAdjacentSpaceToMove.empty() &&
        rightAdjacentSpaceToMove.empty()){
                        This.contents = null;
                        rightAdjacentSpaceToMove.contents = Car;
        }
        Else If (currentSpace.type == Intersection && nextAdjacentSpaceToMove.empty() &&
rightAdjacentSpaceToMove.empty()){
        toTurn = rollProbabilityToTurn();
        If (toTurn){
                This.contents = null;
                rightAdjacentSpaceToMove.contents = Car;
        }
        If(!toTurn){
                This.contents = null;
                nextAdjacentSpaceToMove.contents = Car;
        }
}
}
```

To simulate driving, a road object deletes the object it contains and passes it to another empty object.

If a car is driving towards the end of the simulation, it is deleted.

If a car is on the road and the space ahead of it is empty, it moves forward a space.

If a car is at an intersection and only a single space between its right and forward is available, it moves to that space.

If both spaces are available, a probability roll is generated and moves according to that roll.

Algorithm for RoadEnd Objects

```
        If (this.type == RandomEnd){
                Timer = 0;
                While (simulation is active){
                        Timer++;
                        If (Timer >= TimeBetweenRolls){
                                Timer = 0;
                                Conduct roll with a percentage of (this.probability)
                                If (roll was positive && connecting RoadSegment.empty()){
                                        This.contents = Car;
```

```
                    }
               }
          }

     }
```

If the intersection is a RandomEnd object, a timer is constantly active and advances each clock tick. Once the timer has reached a pre decided point, a probability roll occurs and the timer is reset. If the roll succeeds and the road end is empty, a car is spawned at the end.

```
/////////////////
     If (this.type == TimerEnd){
          OrigCountdown = this.countdown;
          OrigCarCount = this.currentcount;
          While (this.currentcount > 0){
               This.countdown—;
               If (this.countdown <= 0 && connecting RoadSegment.empty()){
                    This.contents = Car;
                    This.countdown = OrigCountdown;
                    this.currentcount—;
               }
          }

     }
```

If the intersection is a TimerEnd object, a timer is constantly active and advances each clock tick. Once the timer has reached a pre decided point and the end is empty, a car is spawned, one is subtracted from the countdown, and the timer is reset. Once the countdown reaches 0, this process ends.