Traffic Simulation: A Case Study for Teaching Object Oriented Design

Viera K. Proulx
Northeastern University
College of Computer Science
Boston, MA 02115, USA
vkp@ccs.neu.edu

0. Abstract

In teaching object oriented design, it is important for students to work on projects that use a variety of design patterns, interaction between objects, and provide the opportunity to explore design options in a realistic setting. Originally, object oriented languages have been designed for use in building simulations. We use a familiar simulation of a traffic through an intersection, controlled by a traffic light as a framework for teaching various aspects of object oriented design. We present this project and show how it illustrates a variety of object oriented design problems.

1. Introduction

When Bjarne Stroustrup first embarked on designing C++, he wanted to build a better tool for programming simulations [2]. His experience with programming in Simula showed him how effective object oriented design is in creating such complex programs. It may be of interest to recall some of his comments:

> *"The class concept allowed me to map my application concepts directly into the language constructs in a direct way that made my code more readable than I had seen in any other language" p. 19.*
>
> *"I ... was very pleasantly surprised by the way the mechanisms of the Simula language became increasingly helpful as the size of the program increased ... the total program acted more like a collection of very small programs than a single large program and was therefore easier to write, comprehend, and debug." p. 20.*

Partial support for this work has been provided by the National Science Foundation Leadership in Laboratory Development , award #DUE- 9650552

If we want our students to become similarly excited about object oriented programming, they need to see examples that illustrate the advantages mentioned by Stroustrup - easier mapping of application concepts into language construct, more readable code, a large program acting like a collection of very small programs.

It is only natural to look at simulations for a source of project ideas. Simulation of traffic on a roadway with an intersection controlled by a traffic light is an excellent choice. Students have a good understanding of the problem domain, can design a limited scope subproblem (e.g., one lane of traffic) as well as extensions (turning traffic, several intersections) which motivates the best students to expand the project. In addition, the state of the system as the simulation proceeds can be displayed easily in a very familiar way. The visual display helps students understand the problems and see their errors. We have been using simple graphics in our introductory courses for a number of years and to create an animation of the system as the traffic moves was a fairly straightforward task.

The programming of the simulation itself is more complex. There are several different objects that both operate independently and interact with each other directly or indirectly. There is the road, the light, and all the cars - all changing their state as the simulation proceeds. Additionally, this simulation allows students to design a whole suite of experiments studying the traffic behavior under different constraints - a different focus altogether from the design of the simulation program. To allow for a suite of such experiments, the user interface design needs to be considered.

In the following sections we will define the problem and identify the necessary objects and their interactions. We will then present our design of this simulation and explain its limitations. We then examine the design lessons to be learned from this project. In the final section we suggest several ways of incorporating this project into coursework at different levels of difficulty.

2. Traffic Simulation - The Problem

The goal of a traffic simulation is to collect data about the traffic flow. In our project we start with modeling one orthogonal intersection controlled by a traffic light. Each roadway leading to and from the intersection will have a fixed length (providing space for a fixed number of cars). The simulation is driven by a timer.

Cars arrive onto the four lanes of traffic leading to the intersection every timer unit with a probability 1:n (n selected by user - suggested value 6). All cars travel at a constant speed of one half car length per timer unit, unless the road ahead is blocked (by another car or by the color of the traffic light). Cars exit the simulation when the front of the car 'falls off the road'.

The duration of the light cycles for the traffic light is selected by the user. Due to the geometry of our design a car needs six timer units to clear the intersection. This determines the duration of yellow light (see Figures 1 and 2).

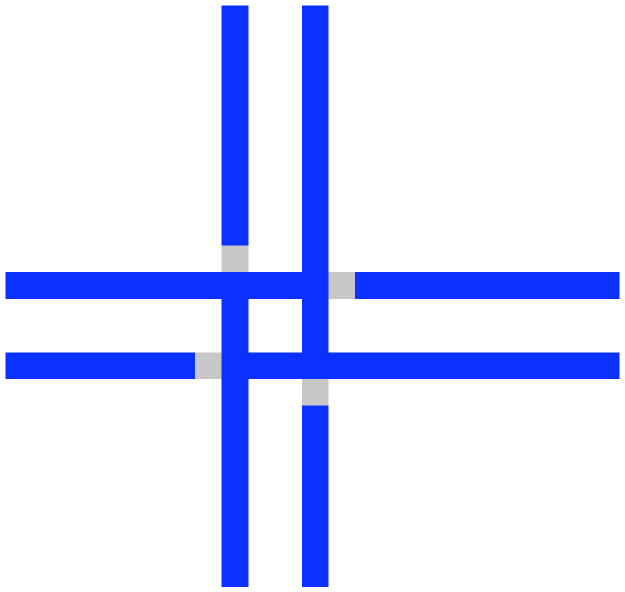
Figure 1. The roadway with the location of places controlled by the traffic light

At this point we do not define the statistics we will collect about the individual cars or about the overall simulation performance, other than counting the number of cars that traversed the entire road in each direction.

During the development process the user input is limited to selecting the traffic light timing, the probability of a car arriving during any given timer unit, and the length of the simulation run. Later, a more complex user interface will allow the user to select the statistics that should be collected and displayed.

We now list the objects involved in the simulation and the actions that these objects perform.

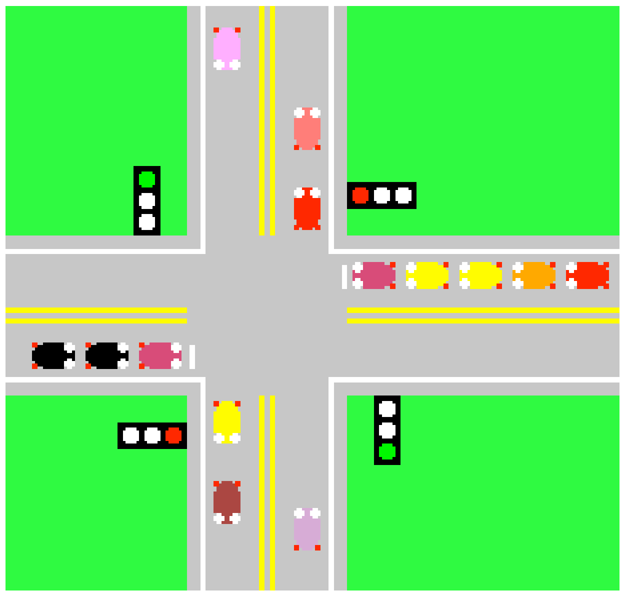
Figure 2. The traffic simulation

Objects:

- road (one lane of traffic - a static structure)
- car (moving in a given direction, currently in a given position)
- collection of cars (every object in the collection needs to be updated during each timer loop)
- traffic light

Actions:

- a mechanism is needed to record whether a given place on the road is free to enter or is blocked
- a car needs to be able to check whether it can move forward
- a car needs to be able to move
- a car needs to detect when it arrives at the end of the road
- the traffic light needs to set its timing
- the traffic light needs to be able to update its status
- the traffic light needs to signal that a particular place on the road is blocked due to its current color
- car collection needs to update all existing cars
- car collection needs to remove exiting cars
- car collection needs to add newly arrived cars

We now describe the details of our design and the reasoning behind each design choice. We then discuss some of the design alternatives as well as the limitations our choices impose on us.

2. Traffic Simulation - Specification of Classes and Their Interactions

Our simulation is controlled by a discrete timer. The granularity of the timer units determines the granularity of the road surface and the atomic car movement increments. Each car occupies two square units of the road surface and can move one unit forward during one timer cycle. These square units (places) form the basis of our simulation. We start by describing the functionality of each object. The design of the display is quite straightforward and we will mention it only briefly.

Place Class

This class is at the heart of the entire program. Each place represents one patch of asphalt on the road. It contains four pointers to other places adjacent to this one. The pointers contain valid links only if travel in that direction is permitted. So, all places in the northbound lane (except for the last one) will have links to their neighbors north of them, but no other links. When we start building intersections, the places where two directions of travel cross will have links to both neighbors. A place can be blocked if it is occupied by a car or if the light ahead is not green.

To display a place (e.g., after a car moved away) we also need to record its physical location in the display. Because the whole display can be treated as a grid specifying the row and column information is sufficient.

Car Class

Each car is represented by an object in the Car class. To distinguish the cars from each other, each is given a color at random (from a list of about 12 colors). The car dimensions are such that each car fits inside two road places. The car object will have two links to the two places it occupies on the road. The car object will also record the direction in which it is facing. This information provides the information needed to display a car or to repaint the background as the car moves.

Additional member data may be added - either to indicate the intent to turn or to collect statistics during the travel.

Member functions will allow us to check if a car can move, to move the car, to check if it is at the end of the road, and, of course, to erase and display the car.

Road Class

This is an interesting class. Each lane of traffic is represented by one object in this class. Each Road object is a linked list of Place objects. The Road object knows its direction, can return a pointer to a specified place on the road, especially to the first place where a car will enter the road.

Traffic Light Class

The TrafficLight class is responsible for timing the light changes and signaling the current color. The signaling of the color is implemented by blocking and clearing the four places in the roads that are just before the entrance to the intersection. This behavior emulates gates at toll booths - a rather drastic method for enforcing the traffic rules. However, it greatly simplifies the object interaction.
To initialize the timing of the traffic light, we need to know the duration of the three light cycles. For the light to function properly, the duration of the red light should be the sum of the green and yellow light in the opposite direction. Knowing that the yellow light has a fixed duration of six cycles the user only needs to specify the duration of green light in two directions (north/south vs. east/west).

The constructor for this class should ask the user to enter the timer settings. The Update function should advance the traffic light timer, display the new state, and block or clear appropriate road places.

CarQueue Class

There is one CarQueue object for each direction of travel. It is a simple queue of car objects. New cars enter at the tail and cars that reach the end are removed from the head. In general, this should be an unstructured collection and we should even be able to update all cars in parallel. However, a queue works well for traffic without turns.

3. Traffic Simulation - The Design Decisions and The Lesson They Teach

We now discuss in detail the various design decisions that needed to be made to make the whole project work. We point out the limitations and suggest alternatives when possible. We also highlight the new concepts students encounter when studying this design.

The decision that each Place object has four links makes the car movement in orthogonal directions easy to implement. It is also possible to make 90° turns in either direction. However, a gradual turn in the road or traffic with a passing lane cannot be easily modeled using this class.

The Place object contains a function **FreeToMove(direction)** that indicates whether there is a link to another Place object in that direction and whether this object is currently marked as **free**. It also has a member function **Next(direction)** that returns a pointer to the next Place object. It is used by the car's **Move** function.

The Car class allow us to make every car into an independent object. Each car should be able to move, erase, and display itself. To do this it needs access to the two Place objects that represent its position. These cannot be member data of the Car object - we need to use only reference to the Place objects that are part of the Road. As the car moves, it unblocks the Place in the rear and blocks the Place ahead. Car can also collect statistics about itself - the total time on the road, the waiting time, the distance traveled, etc. We see that a car designed this way has no knowledge of the world around it other than

its position on the road, the direction of travel, and whether the place ahead is free.
The Road class is a classic example of a static linked structure that is traversed or visited by other objects, but its structure does not change throughout the duration of the simulation.

The Road class and the Place class need to work together here. When building the Road object we need to modify the link member data of Place objects as new ones are added to the road. Once the road is built, these links never change - they are only traversed. The proper design here forces us to do the following:

- there should be no member function in the Place class that changes these links
- the constructor in the Place class should set these links to null
- the Place class should be a friend to Road class, so that the road class has direct access to the link member data
- Road class generates the Place objects and initializes links between them to represent roadways.
- Places that are at the intersection of two roadways are created by the first Road object and the intersecting Road object has to reference the already existing Place object.

This design assures that once the road is built, the links will remain intact throughout the simulation - an important feature (see Figure 3). Such static data structures are important in some system programs and when writing a shared code.

Figure 3. A cartoon

An additional problem arises when two roads cross. Now one Place object that is at the point of intersection will belong to two Road objects. Because in C++ we need to do our own garbage collection, writing a destructor for this linked list requires the use of a reference count. We can only delete Place objects that are not parts of any other Road object.

The TrafficLight object has several interesting features. First, we need to represent the current status of the light and define the transitions between the possible states. Our traffic light design requires that the signal in the two opposite directions (north/south or east/west) are the same at all times. There are at least two methods for recording the current state and computing the new state on each timer update. The first method requires that we store the timing data supplied by the user and every time the light color changes we start a countdown with the appropriate value. The design works, but the implementation is quite messy. Our implementation recognizes the fact that the light cycle repeats after a finite (not very large) number of steps. We represent each possible state of the signals by a pair of letters. GR indicates that the traffic light in the north/south direction if green and that is it red in the east/west direction. The constructor for the TrafficLight object computes the sequence of states during the entire cycle and stores them in an array. It also records the length of the cycle. The current state of the traffic light is then encapsulated in one array index. The update function needs only to increment this array index modulo the cycle length. The key design issue here is computing the state each time vs. storing the state transition diagram data directly. For the purposes of readability, maintenance, and even time efficiency, the second design is clearly a better choice. The additional space requirement for recording the state array is not a problem in this case. Either one of these methods can be easily extended to accommodate light cycles in which the two opposite directions are not synchronized.

The second problem that needs to be addressed is the interaction between the light signal and the cars traveling on the road. To simplify things, the traffic light controls the four Place objects at which the cars enter the intersection. It can block them when the light turns yellow and free then when the light turns green. We may think there is a gate there, like at a toll booth. While this makes it impossible to ignore the traffic rules, it simplifies the object interactions. The discussions of the alternative design options for this interaction will undoubtedly expose a number of interesting problems. A Place object can have a marker indicating we are at a light, maybe even a pointer to the TrafficLight object that the car can then query about its state. Things get more complicated and the design must respect the autonomous nature of each object.

Finally, we examine the alternatives for representing the collection of all cars on the road. The simplest design uses one queue for each traffic lane. Cars enter the queue, traverse the lane, reach the end and exit. The Road object supplies the pointer to the first Place object on the road. When the car reaches the end, the car's first Place pointer becomes null. CarQueue update function performs three tasks:

- moves every car that can move
- removes a car from the front of the queue if it has reached the end of the road
- adds a new car to the queue if "the random number generator says so" and there is a space for this car on the road.

There are no public functions to either insert into the queue or to remove from it - both of these are called internally by the update function.

Another issue is the creation and destruction of Car objects. We elected to use array based implementation of a queue as a circular buffer with Car object as array elements. Instead of creating new Car objects, we use a Reset function when a new car is added to the queue.

If we allow cars to turn, the Car collection needs to be implemented differently. A pointer based linked list is one option. Another one is to allocate memory (array based) to the maximum number of Car objects we may have and mark those objects currently on the road as active. On each update, only the active Cars try to move. New cars are given one of the inactive Car object locations. When a Car object reaches the end of the road, it marks itself inactive.

The display functions can be equipped with a switch that disables the animation for large simulations or if the user wants to conduct a whole series of measurements. Additionally, to display the cars and traffic light in four directions we built a simple vector based toolkit.

4. Traffic Simulation - Additional Features

The design of the traffic simulation described above can be extended in two ways. It is a straightforward extension to add additional roads and intersections, as long as all roads permit travel only in the orthogonal directions. It is only a slightly more difficult to allow cars to turn. A car that plans to turn would need to turn on its turning signal. Each time a car moves, it first tries to move in the direction of the turn, if that is impossible, it moves forward (unless blocked). The design of our roads guarantees that the first step of a turn can always be taken, so we will not miss the turning point because another car may be there. As we mentioned earlier, we need to redesign the way we keep track of all the cars on all roads.

We can add a number of data collecting functions to the car object and extend the simulation to collect this data. We can then design a series of experiments changing the distribution of car arrivals, the timing of the lights, the duration of the simulation. To support such experimentation, we may need to design a user interface and a mechanism for collecting data.

5. Suggestions For Classroom Use

This project can be used in several ways. For students in CS2 we supplied all of the display functions, the complete design document and the definition of all classes. We also supplied a simple driver. Their job was to implement each class. This was a fairly difficult task, but a couple of students got the simulation working. It is possible to downscale the project for introductory courses to model the traffic in only one direction. Another alternative would be to make it a team project with different people responsible for different classes. For a class in object oriented design this can be used as a case study illustrating the numerous design decisions we had to make and especially the mechanisms used to communicate between objects while preserving their autonomy. It could also be either a team project or a capstone project. Depending on the time available and the student's capabilities, we may supply smaller or larger portions of the complete design.

There are also lessons for the instructor arising from this project. It is important to work out a project first before asking students to work on it. A popular textbook (not quoted for obvious reasons) suggests this type of simulation with a graphic display as a simple end of chapter exercise - with no hint about the complex design necessary for a functional implementation. The design choices should be discussed with students and used as examples of the need for different language constructs.

6. Conclusion and Acknowledgments

We started by quoting Bjarne Stroustrup. Looking back, our experience with this design is similar - the code is easier to understand and it feels like a collection of small programs. Extending the functionality affects only a small number of objects in a predictable manner. Students need to see examples of code where good design matters and brings rewards.

In conclusion, I would like to acknowledge the contribution to this project by Richard Rasala - through numerous discussions we had about design issues over the past several years.

References

1. Gamma, R., Helm, R., Johnson, R., and Vlissides, J., *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley, 1995.
2. Proulx, V. K., Rasala, R., and Fell, H., *Foundations of Computer Science: What Are They and How Do We Teach Them?*, SIGCSE Bulletin, June 1996, Vol 28 Special Issue, 42-48.
3. Stroustrup, B., *The Design and Evolution of C++*, Addison Wesley, 1994.