GUI Car Simulation with Processor Depiction

Timothy Dee, Brent Barth

December 18, 2015

Abstract

This report describes a gui simulation created for Cpre 458, Real Time Systems. The simulation depicts a car driving and the state of the processor on the car. The car is shown to be reacting to obstacles in accordance to the tasks being submitting. The tasks are represented visually in a panel which depicts the state of the processor. This paper also discusses the specific implementation of the program. This project design was inspired by the idea of autonomous driving, similar to Google Car.

1 Project Summary

The goal of this project is to learn about the challenges associated with implementing a real-time system. In accordance with this goal, it is necessary to think about real-life situation which can be modeled by simple means. For purposes of this project, the situation must be scalable so a basic working version may be reached relatively swiftly while working toward the more complex desirable implementation. For this reason, we choose to implement a simple version of an autonomous driving system.

The reason such a problem is a good candidate is due several factors. First, the rules are well defined. Most people can easily understand what a car should do when it is on the road. This makes our depiction of the car easy to understand. Second, such a problem is also scalable because an arbitrary number of objects which require some nature of response may be introduced.

The implementation of this system will need to visually represent a car driving on a road. There will also need to be clear obstacles presented to the car. Internally our program will need to use the ideas presented in Real-Time Systems to cause the car to react to the obstacles. This internal state will also need to be visually depicted.

The idea for an autonomous driving simulation was inspired by Google Car. Understanding what challenges need to be overcome overcome to make a system such as this possible is a very interesting prospect. While this implementation project is very far from approaching the complexity level of Google Car, simplified versions of the challenges will remain. Specifically these challenges involve reading data from the ẃorld, deciding what actions to take based on that data, and preforming actions based on those decisions.

2 Objectives and Scope

The scope of this project is rather large. The incorporation of two GUI components adds an amount of difficulty over and above simply a simulation of tasks going though a processor. This extra complexity has a large value though as it puts the problem of implementing a real-time system into a real-life context. Generating tasks based on the state of a GUI is much more visually interesting as well.

The goal will be able to implement everything in a modular way. If this can be done, then it will be easy to get a basic level of functionality while working toward a more complete implementation. A specific example of this might be implementing scheduler interface. Classes which implement that interface could then preform all scheduling operations. In this, a simple scheduler may be implemented swiftly and immediately while a more complex, desirable, scheduler can be implemented once the basic functionality is in place.

Basic functionality for this project will consist of achieving a functional level for all logical components. A component has achieved a functional level if it can take input and provide the appropriate output. A basic functional level for a scheduler then might be defined as the ability to take a list of tasks as parameters and produce a correct schedule. A desirable functional level for the same scheduler might be an efficient algorithm for completing a correct schedule. A desirable functional level for all schedulers might be the implementation of a very interesting scheduler, or a multi-processor scheduler.

The primary objective though is to learn of the challenges involved in the implementation of a real-time system. Put in simple terms, how do you read information from a world which is generating data, analyze that data to decide how to react to it, and cause a reaction based on the decision. The functions of reading and reacting to information are greatly simplified in our virtual world compared to the real world. In the real world there is a vary large amount of noise data, while in our virtual world there are

significantly fewer data sources and these data sources provide clean data. Processing data derived from our virtual world is simplified by virtue of it having less noise when compared to real-world data.

Despite the simplified nature implicit in a virtual world, this project will accomplish its primary objective. The problems faced when dealing with noisy data and specific actuator implementations are problems outside of the scope of this class which delves into the theory of real-time systems. Given these conditions, the fact the the ẃorldís more simple allows the primary objective of this project to be accomplished more easily because this puts the emphasis of the implementation on the theory of a real-time system's operation. Contrast this with a physical implementation where the main concern might be on data acquisition and actuator implementation.

3 Simulation

We seek to implement a system which demonstrates how the ideas from Real Time Systems may be applied to the autonomous driving problem. For this, we choose to implement three distinct logical components. First, we create an entity which keeps track of the state of the ẃorld. This entity will need to show this ẃorldb́ased on the current state, and update the state. It will also need to derive information from the current state of the ẃorldẃhich will enable the processing entity to know if it needs to cause the car to react to an obstacle. Second, there must exist a sort of processing unit which maintains a list of tasks and performs the scheduling. This processing unit takes information from the first entity which maintains the state of the ẃorld, using this information to modify on what element processing will be done. Third, there is a unit which takes in information regarding the state of the processing and shows it visually.

3.1 Simulation Model

Our program code follows a structure having objects with purpose similar to the logical components discussed in the preface of this section. Figure 1 illustrates this layout. In this diagram, lines drawn between two entities represent a control relationship. The MainFrame, for example, controls the Simulate thread. The Simulate thread, in turn, controls three additional threads. Each of these threads is preforming tasks consistent with the logical component they represent. Arrows represent the flow of information.

In the diagram it can be seen that information flows from CarPanelState to SensorData to ProcessingState and finally to ProcessorPanelController. Each of these objects or threads consume the information produced by the previous object or thread. CarPanelState is updated by CarPanelController using logic which is consistent with the laws of physics. This means if an object has a nonzero speed than it will move in accordance with that speed on each update of the panel.

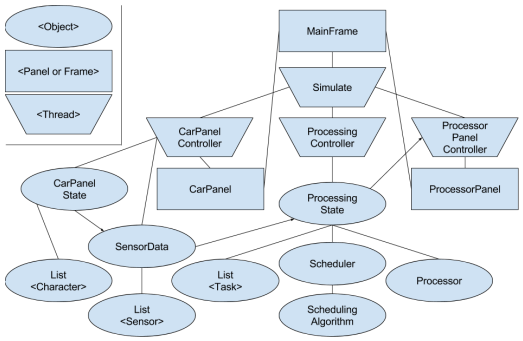

Figure 1: Program Layout

SensorData is the mechanism by which information about the CarPanelState is communicated to ProcessingState. SensorData maintains a list of sensors which interpret the CarPanelState. These Sensors in turn make relevant information about the state accessible. ProcessingState communicates information about the state of the processing to the ProcessorPanelController. ProcessorPanelController in turn uses this to represent the information visually.

The diagram also depicts several threads. Each of these thread will have jurisdiction over one of the logical components of the program. For example, CarPanelController will control how the CarPanel is updated and allow for the acquisition of information about the CarPanelState by other components. Distributing the responsibility in this way makes the responsibilities of each portion of the program very clear. This aids in simplicity of implementation and allows for easier debugging in the sense that when there is an error, the offending code is more easily located.

3.2 Implementation Details

Discussed here are the specific implementation details and program design choices. The overall design philosophy utilized delegates as much processing as possible until it becomes trivial.. This means that components higher in the hierarchy should not need to have any knowledge of how the lower components preform their function. A specific example of this is objects which represent a graphical element knowing how to draw themselves. In this example, the panel which contains the graphical element would ask it to draw itself, this is the delegation of work.

Listing 1: Setup Threads

```
// add the panels to this frame
CarPanel car_panel = new CarPanel(this.width, this.height);
ProcessorPanel processor_panel = new ProcessorPanel();
this.add(car_panel);
this.add(processor_panel);

// start any necessary threads
CarPanelController car_controller = new CarPanelController(car_panel);
ProcessorPanelController processor_controller = new
    ProcessorPanelController(processor_panel);
ProcessingController processing_controller = new
    ProcessingController(scheduling_algorithm, n_processors);

Thread car_controller_thread = new Thread(car_controller);
Thread processor_controller_thread = new Thread(processor_controller);
Thread processing_controller_thread = new Thread(processing_controller);

// start these threads, but they don't act autonamously. Simulate thread
// will use them to perform the simulation.
car_controller_thread.start();
processor_controller_thread.start();
processing_controller_thread.start();

// START SIMULATION THREAD
Simulate simulator = new Simulate(car_controller, processor_controller,
    processing_controller, width, height);
Thread simulator_thread = new Thread(simulator);
simulator_thread.start();
```

3.2.1 MainFrame

This class is used to establish properties of the main window for program simulation. The goal of this class is to arrange the different visual components of the program and begin the simulation. Here is where necessary design decisions about how the responsibilities of the program are delegated. The choice was made to divide the responsibility among three different threads. An additional thread is used to coordinate activity among these other threads. Listing 1 shows the initialization of threads and how they are given to the fourth thread to be managed.

3.2.2 Simulate

Listing 2: Simulate Update Procedure

```
while (true) {
  try {
    // update every 3ms
    Thread.sleep(3);
  } catch (Exception e) {
    e.printStackTrace();
  }

  // provide the SensorData to the
      ProcessingController
  this.processing_controller.
    set_sensor_data(
      this.car_panel_controller.
      get_sensor_data());

  // provide the processing state to the
      processor panel
  this.processor_panel_controller.
    set_processing_state(this.
      processing_controller.get_state());
}
```

This thread has two goals. It's primary purpose is to coordinate activity among the other threads which manage the visual components and the processing. This thread's secondary goal is to set up all of the obstacles and periodic tasks used in the simulation. Periodic tasks, once submitted to the ProcessingController, will be re-injected into the task queue every time their period has expired. The only method by which obstacles may be inserted at this time is by creating them and giving them to the CarPanelController. The original plan for implementing the obstacle generation was to create a thread which would insert obstacles occasionally. This is something we were not able to implement due to time constraints.

Listing 2 describes the update procedure. The SensorData construct is taken from the CarPanelController and given to the ProcessingController thread to be utilized by the currently running periodic and aperiodic tasks. The processing state is taken from the ProcessingController and given to the ProcessorPanelController to be drawn. This update procedure happens at a frequency greater than the update procedures in the other threads. The purpose of this increased frequency is to attempt to ensure the data the ProcessingController and ProcessingPanelController are using are not stale. This is what is meant by c̈oordinate activity among the other threads.̈

3.2.3 CarPanelController

Listing 3: Panel Update Procedure

```
private void update_panel() {
  // update the targets if necessary
  update_target_state();

  // call methods to perform various
      specific updates
  move_car();
  move_obstacles();
  move_other_cars();
  move_signs();
  move_road();

  // cause the panel and its children to
      repaint
  this.car_panel.repaint();
  this.car_panel.revalidate();
}
```

Listing 3 depicts the method called within CarPanelController as the update procedure. That is, this function runs approximately every 10 milliseconds. This class maintains the real state of the objects drawn to the panel depicting the car. This class also maintains another state which describes the intended position or goal for the objects. We call this state the target state. Most methods in the update procedure update the target state in some way.

The first method, update_target_state, performs any updates to the target state which are not related to any objects on screen. The next set of methods update the target state based on the speed and current position of the objects on screen. The general rule is if the target state is different from the real state, then we move the real state one closer to the target state. In this way, we maintain incremental movement of the objects on screen. This ensures the objects move smoothly.

3.2.4 ProcessingController

Listing 4: Processing Controller Update Procedure

```
while (true) {
  try {
    // update every 10 ms (100 fps)
```

```
    Thread.sleep(10);
   } catch (Exception e) {
    e.printStackTrace();
   }

   // ask the state to update itself
   processing_state.update(1L);
}
```

Listing 4 depicts the update process for processing controller. This method calls the update process of the processing state every 10 milliseconds. This is necessary to ensure the processing state is being updated at a similar rate as the CarPanelState. The processing_state.update method preforms updates in an incremental way based on the amount of time which has elapsed since the last update. This is why we provide the argument 1L. This will cause the state to update as if 1 unit of time has passed.

The purpose of the ProcessingController is to manage all aspects of the processing. This means deciding how much to update the processing_state, and modifying the processing state based on what is going on in the rest of the program. ProcessingController is also responsible for updating the sensor data in the processing state, and for introducing new tasks into the processing state.

3.2.5 ProcessorPanelController

Listing 5: Processing Panel Update Procedure

```
while (true) {
 try {
  // update every 10 ms (100 fps)
  Thread.sleep(10);
 } catch (Exception e) {
  e.printStackTrace();
 }

 set_processing_state(this.
  processor_panel.getState());

 // cause the processor panel to be
     repainted
 processor_panel.repaint();
 processor_panel.revalidate();
}
```

Listing 5 shows the update procedure for the ProcessingPanelController. This ProcessingPanelController is responsible for managing the updates of the ProcessorPanel. It first updates the state of the ProcessingPanel. Then it calls the necessary methods to have the panel and its children repaint themselves. This class also mediates interactions between the ProcessorPanel and the rest of the program.

3.2.6 CarPanel

Listing 6: Ask Signs to Draw Themselves

```
for (Sign c : state.signs) {
 c.draw(g);
}
```

Listing 6 illustrates the design philosophy followed by the program as a whole. The CarPanel maintains a state. This state is set by the CarPanelController on each update. From the state the CarPanel asks each of the visual components to draw themselves. In this way, the task of drawing the objects on the screen is delegated as far as possible. It also allows the tasks of deciding where to draw the the objects and the actual drawing of the objects to be separated. This aids in the simplicity of the code.

More specific the separation of tasks is as follows. First the CarPanelController decides where all of the elements need to be drawn on the screen. The CarPanelController then hands this information off to the CarPanel which asks all of the items to draw themselves. The classes created to represent the items handle the actual drawing of the item. In addition to keeping the code simple, this allows all information relevant to an item to be kept in the same class. The Sign class, for example, contains information about what should be on the Sign and where the Sign should be as well as information about how to draw the Sign to the screen.

3.2.7 CarPanelState

Listing 7: Car Panel State maintains all objects drawn on the screen

```
public MainCar main_car;
public Road road;

public ArrayList<Cone> obstacles;
public ArrayList<Car> other_cars;
public ArrayList<Sign> signs;
```

Listing 7 depicts what is maintained in CarPanelState. CarPanelState acts like a data structure to hold all of the Characters which will be drawn on the screen. The CarPanelState is the programmatic representation of what will be drawn on the CarPanel. The CarPanelState has its information updated by other classes. The only processing provided by CarPanelState is a reset function.

3.2.8 Character

Listing 8: Sample Implementation of a Character

```
public void draw(Graphics g) {
 // draw the body of the car
```

```
  g.setColor(this.color);
  g.fillRect(this.x_pos, this.y_pos,
      this.width, this.height);

  if (Facing.RIGHT == this.facing) {
    // draw the windshield
    g.setColor(Color.black);
    g.fillRect(this.x_pos + this.width / 2
        + this.width / 16, this.y_pos +
        this.height / 10, this.width / 3,
       this.height * 4 / 5);
  } else {
    // draw the windshield
    g.setColor(Color.black);
    g.fillRect(this.x_pos + this.width /
        16, this.y_pos + this.height / 10,
        this.width / 3,
       this.height * 4 / 5);
  }
}
```

The Character class is an abstract class which is the parent of all items which will be drawn on the panels. The idea of the Character class is to take care of all the common functionality among the items which will be drawn to the screen. This common functionality includes a constructor and a copy constructor which provide an easy way to set all of the parameters such as: x position, y position, color, height, and width. The abstract method provided by Character is the draw(Graphics g) method. Any class that extends the Character class will be responsible for creating an implementation of this method.

Listing 8 describes how a class might override the draw method of the Character class. This listing was taken from the implementation of Car which represents the blue car which is driving on the road as well as other red cars which are driving in the opposing direction to the blue car. The draw method above draws a rectangle for the car at the x postion and y postion made available by virtue of the fact that Car extends Character. It then uses the facing enumerated class to decide on which side the windshield should be drawn

3.2.9 SensorData

Listing 9: Sensor Data maintains a number of sensors

```
public StopSignSensor stop_sign_sensor;
public SpeedSignSensor speed_sign_sensor;
public OtherCarSensor other_car_sensor;
public ConeSensor cone_sensor;
```

Listing 9 shows the objects maintained within sensor data. Similar to CarPanelState, this class acts like a data structure. The important reason to have a SensorData class, instead of simply using a number of variables to hold the values SensorData stores is due to simplification of code. The SensorData constructor takes the CarPanelState as a parameter. It then uses this state information when constructing the sensors. If not for the Sensor data class, this would need to be done elsewhere, and would almost certainly meed to be done at the same time wherever it was completed. SensorData is the mechanism by which state information about the CarPanel is transmitted to the ProcessingController.

3.2.10 Sensor

The Sensor class itself is an abstract class which knows how to preform tasks common among all sensors. All Sensors have a concept of range, or the maximum distance in pixels which the sensor will look for other objects. It follows from this that there should be a way for each sensor to tell whether an object is in range or not. The sensor class provides the iswithin_range() function and the compute_distance function for this purpose.

In Listing 10 a sample child of the Sensor class is shown. This class overrides the compute() abstract method of the Sensor class. The last method of the Sensor constructor calls the compute() method. In this way, any values meant to bee computed by a sensor will be computed when the Sensor is constructed. All of the sensors compute values as a function of the CarPanelState and provide those values in their public variables. Sensors exist for Speed Signs, Stop Signs, Cones, and OtherCars.

3.2.11 ProcessingState

Listing 11 describes the method used to update the processors. Updates are preformed incrementally as a function of the number passed in as a time argument. For each unit of time, this method will check if a task has completed in any of the processors. If a task has finished, we run the scheduler to see what should be the next task executed by the processor. If there are no tasks waiting, we insert a dummy task, otherwise we place the task with highest priority into the processor. We choose to keep the scheduler independent of this method to allow new scheduler types to be implemented modularly.

3.2.12 Task

Listing 12: Tasks know how to preform their actions

```
// describes the nature of the task
public enum Nature {
  PERIODIC, APERIODIC;
}

// says what action should be conducted
    upon task completion
```

Listing 10: Implementation of Sensor

```
public class StopSignSensor extends Sensor {
  public ArrayList<Sign> signs;
  public ArrayList<Double> distances;

  public StopSignSensor(CarPanelState state) {
    super(state);
  }

  @Override
  protected void compute() {
    this.signs = new ArrayList<Sign>();
    this.distances = new ArrayList<Double>();

    for (Sign sign : this.car_panel_state.signs) {
      if (sign.type == SignType.STOP && is_within_range(this.car_panel_state.main_car,
          sign)) {
        // if within range of sensor
        signs.add(sign);

        // add in parallel the distance to the sign
        distances.add(compute_distance(this.car_panel_state.main_car, sign));
      }
    }
  }
}
```

```
public enum Action {
  NONE, SET_CAR_SPEED, MOVE_UP_CAR,
      MOVE_DOWN_CAR, READ_CONE_SENSOR,
      READ_OTHER_CAR_SENSOR,
      READ_SPEED_SIGN_SENSOR,
      READ_STOP_SIGN_SENSOR;
}

public void preform_action();
```

Listing 12 describes the basic properties of a task. Tasks maintain information about whether they are periodic or aperiodic. This information is used when the task is removed from the processing to determine whether or not to create a task with similar properties at the time its period would have expired. Based on their action, tasks will choose to do different things in the preform_action method. Tasks may also have no effect if an action of NONE is specified.

Tasks take one of two forms. Tasks will either read a sensor or preform an action which affects the CarPanelState. Whenever a task reads a sensor, it will start any appropriate aperiodic tasks. This could mean, for example, changing the speed of the MainCar based on a speed limit sign or bringing the car to a stop for a stop sign. Tasks may affect the CarPanelState by setting the speed of the MainCar or telling the MainCar to move up or down.

3.2.13 Scheduler

Listing 13: The scheduler uses a scheduling algorithm to choose the order of tasks

```
/**
 * returns the task set in scheduled order.
 */
public ArrayList<List<Task>>
    schedule(List<Task> tasks, int
    n_processors) {
  ArrayList<Task> array_list_tasks = new
      ArrayList<Task>(tasks);

  // discard tasks which will already
      missed their deadline
  discard_late_tasks(array_list_tasks);

  // if the schedule is feasible, schedule
      it. Otherwise return null.
  if (this.scheduling_algorithm.
    is_feasible_schedule(tasks,
        n_processors)) {
    return (ArrayList<List<Task>>)
        this.scheduling_algorithm.
      schedule(array_list_tasks,
          n_processors);
  } else {
```

Listing 11: Scheduler call and task queue manipulation

```
private void update_processors(long time) {
  // preform updates in an incremental fashion
  for (int i = 0; i < time; i++) {
    total_time++;
    add_periodic_tasks();

    // if a task has completed, run the scheduler at the point it completes.
    // remember the only tasks who's computation time is decreasing are the ones in the
        processors.
    for (Processor p : processors) {
      // first, decrement the computation time of the thing in this processor
      p.task.computation_time_remaining--;

      if (p.task.computation_time_remaining <= 0) {
        // preform the complete action of the task when it is finished
        p.task.preform_action();

        // if a task has completed, we might need to run the scheduler
        run_scheduler();

        // if the task has finished, get the next thing out of the queue
        if (p.task_queue.isEmpty() == false) {
          // get the next task in the queuue
          p.task = p.task_queue.get(0);
          // take out of processor queue
          p.task.taskBlock.inProcessorQueue = false;

          if (this.processorTasks != null) {
            if (this.processorTasks.size() > 0) {
              this.processorTasks.remove(0);
            }
          }

          if (this.processorTasks != null) {
            if (this.processorTasks.size() == 0) {
              // put in processor
              this.processorTasks.add(p.task.taskBlock);
              this.processorTasks.get(0).inProcessorQueue = false;
              this.processorTasks.get(0).inProcessor = true;

            }
          }

          // remove task from the processor queue for gui
          if (this.processorQueueTasks.size() > 0) {
            this.processorQueueTasks.get(0).inProcessorQueue = false;
            this.processorQueueTasks.remove(0);
          }

          // remove this task from the queue
          p.task_queue.remove(0);
        } else {
          // insert a dummy task
          p.task = new Task(1, 0, 0, Nature.APERIODIC, Action.NONE, null, null, 0);
} } } } }
```

```
    System.out.println("OVERLOAD");
    // if there is no feasible schedule,
        use the overload scheduling
    // algorithm
    return (ArrayList<List<Task>>)
        this.overload_scheduling_algorithm.
      schedule(array_list_tasks,
          n_processors);
  }
}
```

Listing 13 describes the Scheduler class which may be instantiated to create a scheduler with a SchedulingAlgorithm given in the constructor. This Class is designed to be able to handle any number of processors with number of processors being determined by parameter n_processors. The schedule method describes a list of lists should be returned. In this, each list is to refer to the tasks which should be scheduled on one processor. The interface will return null if the list of tasks provided as a parameter is not schedule-able.

Scheduler has two SchedulingAlgorithms. One SchedulingAlgorithm is used when the processor utilization is under 100%. The other SchedulingAlgorithm is used when the processor utilization is over 100%. Scheduler will use the feasibility check of the normal case scheduling algorithm to determine whether whether the task set is schedule-able. If it is schedule-able, the normal case scheduling algorithm will be used. If, however, the feasibility check returns false, Scheduler will use the scheduling algorithm for the overload condition.

We decided to implement the scheduler in this way because it allows for easy additions of new SchedulingAlgorithms. All we need to do to have our program use a new scheduling algorithm is provide the scheduler a new class which extends the SchedulingAlgorithm interface. This will fulfill our initial requirement that a working version of the code should be achievable with relative ease.

3.2.14 SchedulingAlgorithm

Listing 14: Scheduling Algorithm interface

```
public interface SchedulingAlgorithm {
  /**
   * returns a schedule of all tasks for n
       processors. Each processor has a
   * different list in the task set.
   *
   * @param tasks
   * @return
   */
  public List<List<Task>>
      schedule(List<Task> tasks, int
      n_processors);

  public boolean
      is_feasible_schedule(List<Task>
      tasks, int n_processors);
}
```

Listing 14 show the interface SchedulingAlgorithm. A class which implements this interface can be provided to Scheduler in order to determine how the schedule will schedule incoming tasks. When the scheduler is asked to schedule something, it will call the schedule method of the scheduling algorithm it was provided. SchedulingAlgorithms are also expected to implement a schedule-ability check. This is done by implementing the is_feasible_schedule() method. This check is necessary for the processor to know when it should use a scheduling algorithm for the overload case. In this way, any number of SchedulingAlgorithms might be implemented.

3.2.15 Processor

Listing 15: Values maintained in Processor

```
public Task task;
public List<Task> task_queue;
```

Listing 15 shows the information maintained in the processor class. This information is used by the processor controller to manage the processor. In this way, processor is essentially a data structure. Importantly though, processor is stored in the ProcessingState class. This fact means that processor is provided to the ProcessorPanel to be visually depicted on the screen. This allows the currently running task to be shown inside the processor.

3.2.16 ProcessorPanel

Listing 16: Update process of ProcessorPanel

```
protected void paintComponent(Graphics g) {
  super.paintComponent(g);

  drawSchedulerQueueTasks(g);
  drawTaskTables(g);
  drawLabels(g);
  draw_signs(g);
}
```

Listing 16 illustrates the functions of the ProcessorPanel. The ProcessorPanel is the Panel which visually represents the state of the processing in the lower half of the display screen. It shows how ProcessorPanel organizes the drawing of its elements into specific methods. Each of these methods manages one of the items which will be drawn to the screen. Each of these methods follows a similar design pattern to the methods in CarPanel. Each method

asks a set of Characters to draw themselves to the screen.

3.3 Implemented Scheduling Algorithms

3.3.1 EDF

Listing 17: EDF scheduler implementation

```
// find the earliest deadline, put it first
for (int i = 0; i <
    scheduled_tasks.size(); i++) {
  for (int j = i + 1; j <
      scheduled_tasks.size(); j++) {
    // sorted smallest thing first
    if (scheduled_tasks.get(i).deadline >
        scheduled_tasks.get(j).deadline) {
      Task temp = scheduled_tasks.get(i);
      scheduled_tasks.set(i,
          scheduled_tasks.get(j));
      scheduled_tasks.set(j, temp);
    }
  }
}
```

Listing 17 shows the EDF implementation and how we schedule the tasks in the scheduler queue. We go through all the tasks in the scheduler queue to find the task with the earliest finishing deadline and set that as the next available task for the processor. This allows for a quick and clean implementation of the EDF scheduling algorithm.

3.3.2 HVDF

Listing 18: HVDF scheduler implementation

```
// find the largest value task, put it
    first
for (int i = 0; i <
    scheduled_tasks.size(); i++) {
  for (int j = i + 1; j <
      scheduled_tasks.size(); j++) {
    // sort the largest value thing first
    if
        (value_density(scheduled_tasks.get(i))
        <
        value_density(scheduled_tasks.get(j)))
        {
      Task temp = scheduled_tasks.get(i);
      scheduled_tasks.set(i,
          scheduled_tasks.get(j));
      scheduled_tasks.set(j, temp);
    }
  }
}
```

Listing **??** shows the HVDF implementation and how we schedule tasks when there needs to be overload handling. We go through all the tasks in the scheduler queue to find the task with highest value density and set that task as the next task in the processor queue. We use the value_density function to determine a tasks density, and then compare it with the other tasks to find the highest density value.

3.4 Outputs

The output from our GUI Car Simulation is divided into two different parts. One section depicts the Car Simulation itself, which displays objects and obstacles that the car must react to. The second section displays the scheduler, processor, and how tasks are processed by the car as a real-time system. The overall goal is to display both the car and the scheduler, processor working together to give a visual depiction of how this real-time system could be implemented.

The first panel which corresponds to the actual car has many different objects to show. In figure 2, there are many different objects drawn to the screen. The first being the blue car, which is the main car that is simulating the real-time system and will react to stimuli on the panel. The cars speed is the number in black in upper left hand corner. The stimuli the car reacts to can range from a cone (shown below in orange), other cars (shown below as red car), speed limit signs (shown below in white rectangle), car lanes / off road grass (shown below), and lastly stop signs (not pictured). When the cars sensors recognizes these objects they create an aperiodic task. When this task is executed by the processor, the car will react with the action the aperiodic task was created to handle. Thus creating a real-time car simulation that periodically searches for stimuli and creates aperiodic tasks to react to them.

The second panel corresponds to the scheduler / processor to which the tasks created by the cars sensors are handled. Figure 2 illustrates the four different sections of the scheduler / processor. The first section is the scheduler queue which consists of both periodic and aperiodic tasks. The periodic tasks are displayed as the red squares (as shown below), and the aperiodic tasks are displayed as blue squares (not pictured). The next step in the process is for these tasks to be scheduled which is represented by the black and red circle labeled Scheduler (as shown below). After the tasks have been scheduled, the next task to be put in the processor in displayed in the processor queue (as shown below with no task). Once the processor is empty it will take the task from the processor queue and execute it in the processor (as shown below). This is the process for how the scheduler / processor is displayed in our GUI Car Simulation.

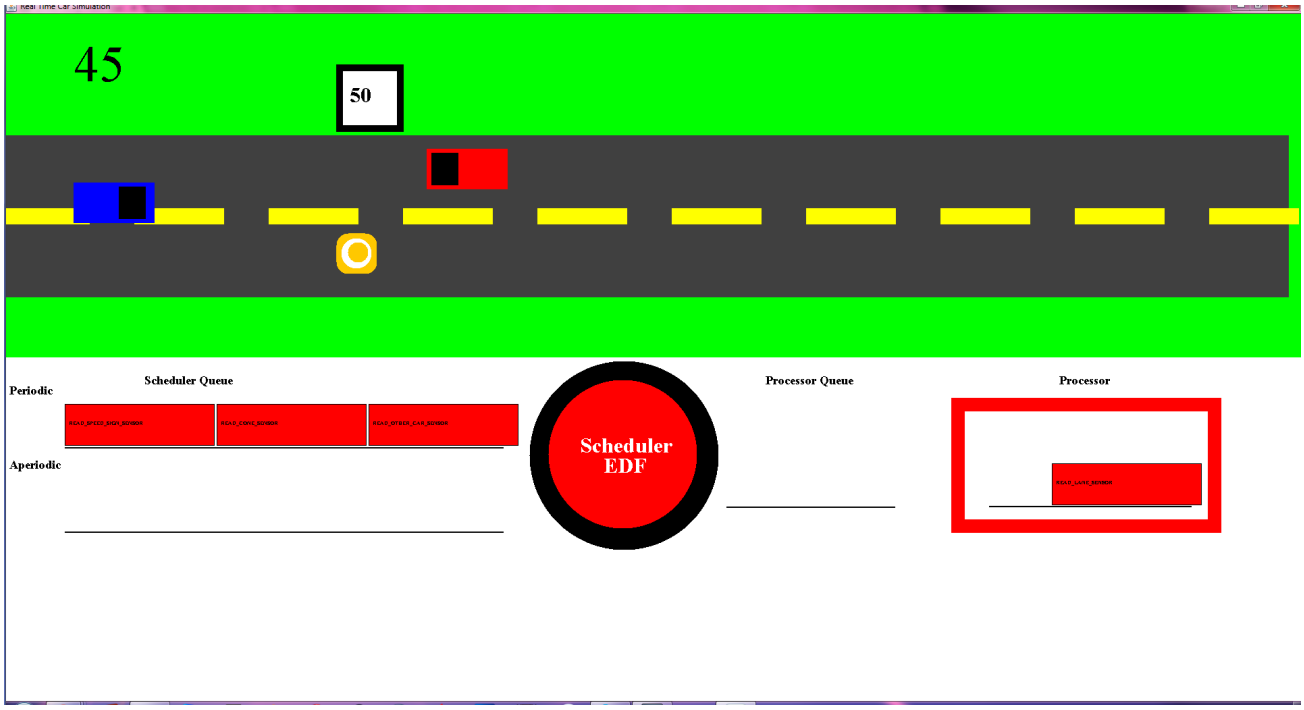

Figure 2: This image was taken during a simulation. It depicts the objects displayed on the CarPanel (top) and ProcessorPanel (bottom).

4 Conclusion

In conclusion to this project, we have learned about many different topics covered in real-time systems. Tasks, schedulers, sensors, and processing were all different categories showcased within our project. Each step of the project held different challenges to overcome, as we tried to create an environment which would allow us to implement systems similar to those discussed in class. The GUI Car Simulation really brought home the idea of how a real-time system should work, as well as the challenges of dividing tasks, prioritizing tasks, and executing them in a timely manner. The implementation of a GUI Car Simulation has many different parts and the implementation took great design and detail. The design we used made it easy to describe and display the tasks we felt were the foundation of an autonomous car. We felt this made our project special in the sense that we depicted not only what the car sees but also what the real-time system executing behind the scenes. Being able to display both of these scenarios simultaneously allowed us to explore the inner workings of how a real-time system works and should be implemented.

4.1 Synopsis

The synopsis from our project is that we are glad we implemented both EDF and HVDF. This allowed us to implement overload handling for when tasks are not scheduleable by the EDF scheduling algorithm as well as for our scheduling to be handled even in the most non-ideal situations. Another reason for this implementation is that in the future we could have a random arrangement of obstacles and let the car react based on tasks created by the obstacles. The overload handling would be able to handle difficulties with multiple aperiodic and periodic tasks in the queue for processing. Based on our project implementation of the schedulers, we could easily add different scheduling algorithms.

We also only decided to implement a uniprocessor system. The reason for this is because we did not know the whole outlook for the project and how long it would take to complete. Due to the way we implemented our real-time system simulation, we could have easily expanded it into a multiprocessor system. The changes we would have gone with consisted of different scheduling algorithms, different tasks with other actions, and lastly a changed GUI to show all of the changes that would be presented with implementing another/more processor(s).

4.2 Learning

The primary objective of this project was to learn about the challenges involved in implementing a real time system. It seems this goal was accomplished. We feel that we have

a much more complete understanding of the ideas in this course. It is one thing to speak about theory. It is an entirely different thing to implement the theory. There was also learning concerning the specific problems involved in implementing a GUI simulation. Something which is seemingly trivial, such as a car driving on a road, can present interesting problems making it exceptionally challenging.

Learning in the context of real-time systems was benefited by creating something and seeing what issues present themselves in practice allows a person to understand the motivation behind doing things in a particular way. For instance, a person might ask, Ẅhy not just complete the tasks in the order they arrive?̈. Before doing this project, I would have probably answered in terms of the theory of real-time system. Such an answer might involve why we must maintain priorities for tasks so these tasks meet their deadline. After doing this project, I can now answer in the context of a real-life example. I can say *why* tasks must meet their deadlines and give illustrative examples of what might happen should a task miss it's deadline.

Throughout the semester, we spoke of many solutions in real-time systems. For us, the primary knowledge gained from this project was the motivation behind the development of these solutions. For example, we talked about different algorithms for combined scheduling in class. Some algorithms provide better response time for aperiodic tasks, but make trade offs in other areas. What we gained from having done this project is an understanding of why better response time for aperiodic tasks might be desirable in the first place.

There was also significant knowledge acquisition in our ability to create GUI simulation simulations. There are many interesting problems related to attempting to visually realize data. One of the first challenges we faced in creating our solution was designing the organizational structure in a modular way which would allow for the addition of new components. To overcome this challenge we decided to delegate work as far as possible. This way any additional objects, such as a new obstacle or sign, would not have to worry about any of the structure above them. In some sense the goal was to maintain a separation of responsibility. In this paradigm delegating work as far as possible was the algorithm used to accomplish that.

Another significant issue in creating a GUI simulation is fluid motion. To enable objects to move fluidly on the screen is not trivial; doing this requires the state of the objects to be updated in a very specific way. CarPanelController initiates an update of CarPanel every 10 milliseconds. This equates to roughly 100 frames per second. Of course it will be less because the frequency of updates in reality is $fps = \frac{1000ms}{1s} \times \frac{1frame}{10ms+n}$ where n is the time in milliseconds taken by the code that describes what is to be drawn on the panel. If the screen is not updated frequently enough, then the movement will appear choppy. This is why it is necessary to update so many times. Likely the screen is only updated around 60 times per second due to the refresh rate of most monitors, having our program preform the update around 100 times per second allows us to always meet this number of frames.

Another component of the fluid moment problem, apart from when to draw to the screen, is deciding what to draw to the screen. We found that if you try to move objects more than one or two pixels at a time then the objects will appear choppy. This gave rise to a really neat way of updating the CarPanelState from which the objects in the CarPanel are drawn. CarPanelController maintains two states. One state is the target state which is updated by external sources, namely tasks, when these sources want to tell the car to move somewhere on the screen. The other state is the real state that is given to CarPanel to be drawn. Every time CarPanelController updates, the real state is moved one step closer to the target state. For instance, if the y position of the main car in the real state is greater than the y position of the main car in the target state, than the y position of the car in the real state will be decremented before being given to the CarPanel to be drawn. In this way, objects are only moved one or two pixels at a time, assuming the refresh rate of the monitor running this program is above $50hz$.

4.3 Suggestions

It is good that the project is open-ended. This gave us an opportunity to learn in a way which is interesting to us. I felt a lot more motivated to do this project then I would have if it was a normal well-defined, highly-structured project. It is much nicer to learn this way in my opinion.

I think it would be better if the project began earlier in the semester. Having the start of the project be less than three weeks before dead week put a huge strain on time at this point in the semester. I understand that it doesn't make sense to start the project before a significant amount of material has been covered. Potentially making the report requirement slightly less substantial would help.