GUI Car Simulation with Processor Depiction

Timothy Dee, Brent Barth

December 3, 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.

2 Introduction

3 Objectives and Scope

4 Solution Approach

4.1 Algorithms

4.2 Solution Sample

5 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 ẃorld́based on the current state, and update the state. It will also need to derive information from the current state of the ẃorld́which 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.

5.1 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

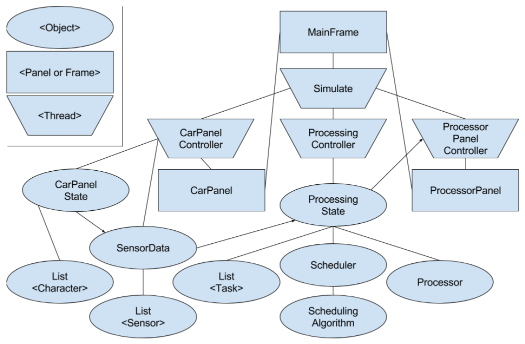

Figure 1: Program Layout

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.

5.2 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.

5.2.1 MainFrame

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();
```

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 addition thread is used to coordinate activity between these threads. Listing 1 shows the initialization of threads and how they are given to the fourth thread to be managed.

5.2.2 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_pane

  // 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 between the 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. 2

5.2.3 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();
}
```

3

5.2.4 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);
}
```

4 depicts the update process for processing controller

5.2.5 ProcessorPanelController

```
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();
}
```

5.2.5

5.2.6 CarPanel

Listing 5: Ask Signs to Draw Themselves

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

5 illustrates the design philosophy. This shows how components are asked to draw themselves.

5.2.7 CarPanelState

```
public MainCar main_car;
public Road road;

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

5.2.7

5.2.8 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);
  }
}
```

5.2.8

5.2.9 SensorData

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

5.2.9

5.2.10 Sensor

Listing 6: 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));
      }
    }
  }
}
```

6

5.2.11 ProcessingState

```
private void update_processors(long time) {
  // preform updates in an incremental
      fashion
  for (int i = 0; i < time; i++) {
    // 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) {
        // periodic tasks need to be added
            back
        if (p.task.nature ==
            Task.Nature.PERIODIC) {
          // add the same task back in if it
              is periodic
          this.scheduler_task_queue.add(new
              Task(p.task.computation_time_origional,
              p.task.period,
             p.task.deadline, p.task.nature,
                 p.task.action,
                 p.task.processing_controller,
             p.task.car_panel_controller,
                 p.task.set_point));
        }

        // preform the complete action of
            the task when it is
        // finished
        p.task.preform_action();

        // if the task has finished, get the
            next thing out of the
        // queue
        // set the current processor task to
            the smallest thing in
        // the queue
        if (p.task_queue.isEmpty() == false)
            {
          // get the next task in the queuue
          p.task = p.task_queue.get(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);
        }
      }
    }
  }
}
```

5.2.11

5.2.12 Task

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

5.2.12

5.2.13 Scheduler

```
/**
 * returns the task set in scheduled
     order. null if not schedulable.
 */
public ArrayList<List<Task>>
    schedule(List<Task> tasks, int
    n_processors) {
  ArrayList<Task> array_list_tasks = new
      ArrayList<Task>(tasks);

  return
      this.scheduling_algorithm.schedule(
    array_list_tasks, n_processors);
}
```

5.2.13

5.2.14 SchedulingAlgorithm

```
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);
}
```

5.2.14

5.2.15 Processor

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

5.2.15

5.2.16 ProcessorPanel

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

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

5.2.16

5.3 Outputs

6 Conclusion

6.1 Synopsis

6.2 Learning

6.3 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.