GUI Car Simulation with Processor Depiction

Timothy Dee, Brent Barth

December 4, 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 world. This entity will need to show this world based on the current state, and update the state. It will also need to derive information from the current state of the world which 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 world, 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
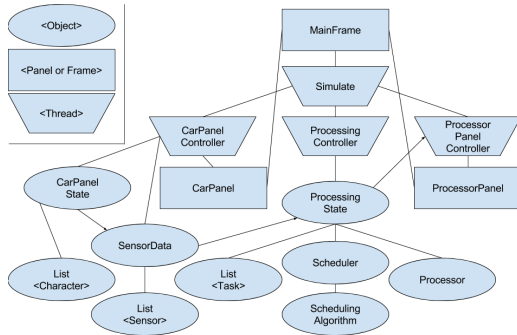
Figure 1: Program Layout

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

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

```
1  while (true) {
```

```
2   try {
3     // update every 3ms
4     Thread.sleep(3);
5   } catch (Exception e) {
6     e.printStackTrace();
7   }
8
9   // provide the SensorData to the
        ProcessingController
10  this.processing_controller.
11    set_sensor_data(
12      this.car_panel_controller.
13      get_sensor_data());
14
15  // provide the processing state to the
        processor panel
16  this.processor_panel_controller.
17    set_processing_state(this.processing_controller.
18      get_state());
19 }
```

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

```
1  private void update_panel() {
2    // update the targets if necessary
3    update_target_state();
4
5    // call methods to perform various
        specific updates
6    move_car();
7    move_obstacles();
8    move_other_cars();
9    move_signs();
10   move_road();
11
12   // cause the panel and its children to
        repaint
13   this.car_panel.repaint();
14   this.car_panel.revalidate();
15 }
```

    3

5.2.4 ProcessingController

Listing 4: Processing Controller Update Procedure

```
1  while (true) {
2    try {
```

Listing 1: Setup Threads

```
1  // add the panels to this frame
2  CarPanel car_panel = new CarPanel(this.width, this.height);
3  ProcessorPanel processor_panel = new ProcessorPanel();
4  this.add(car_panel);
5  this.add(processor_panel);
6
7  // start any necessary threads
8  CarPanelController car_controller = new CarPanelController(car_panel);
9  ProcessorPanelController processor_controller = new
       ProcessorPanelController(processor_panel);
10 ProcessingController processing_controller = new
       ProcessingController(scheduling_algorithm, n_processors);
11
12 Thread car_controller_thread = new Thread(car_controller);
13 Thread processor_controller_thread = new Thread(processor_controller);
14 Thread processing_controller_thread = new Thread(processing_controller);
15
16 // start these threads, but they don't act autonamously. Simulate thread
17 // will use them to perform the simulation.
18 car_controller_thread.start();
19 processor_controller_thread.start();
20 processing_controller_thread.start();
21
22 // START SIMULATION THREAD
23 Simulate simulator = new Simulate(car_controller, processor_controller,
       processing_controller, width, height);
24 Thread simulator_thread = new Thread(simulator);
25 simulator_thread.start();
```

```
3      // update every 10 ms (100 fps)
4      Thread.sleep(10);
5    } catch (Exception e) {
6      e.printStackTrace();
7    }
8
9    // ask the state to update itself
10   processing_state.update(1L);
11 }
```

4 depicts the update process for processing controller

5.2.5 ProcessorPanelController

Listing 5: Processing Panel Update Procedure

```
1  while (true) {
2    try {
3      // update every 10 ms (100 fps)
4      Thread.sleep(10);
5    } catch (Exception e) {
6      e.printStackTrace();
7    }
8
9    set_processing_state(this.processor_panel.getState());
10
11   // cause the processor panel to be
          repainted
12   processor_panel.repaint();
13   processor_panel.revalidate();
14 }
```

5

5.2.6 CarPanel

Listing 6: Ask Signs to Draw Themselves

```
1  for (Sign c : state.signs) {
2    c.draw(g);
3  }
```

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

5.2.7 CarPanelState

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

```
1  public MainCar main_car;
2  public Road road;
3
4  public ArrayList<Cone> obstacles;
5  public ArrayList<Car> other_cars;
6  public ArrayList<Sign> signs;
```

7

5.2.8 Character

Listing 8: Sample Implementation of a Character

```
1  public void draw(Graphics g) {
2    // draw the body of the car
3    g.setColor(this.color);
4    g.fillRect(this.x_pos, this.y_pos,
          this.width, this.height);
5
6    if (Facing.RIGHT == this.facing) {
7      // draw the windshield
8      g.setColor(Color.black);
9      g.fillRect(this.x_pos + this.width / 2
          + this.width / 16, this.y_pos +
            this.height / 10, this.width / 3,
          this.height * 4 / 5);
10   } else {
11     // draw the windshield
12     g.setColor(Color.black);
13     g.fillRect(this.x_pos + this.width /
            16, this.y_pos + this.height / 10,
            this.width / 3,
14         this.height * 4 / 5);
15   }
16 }
```

8

5.2.9 SensorData

Listing 9: Sensor Data maintains a number of sensors

```
1  public StopSignSensor stop_sign_sensor;
2  public SpeedSignSensor speed_sign_sensor;
3  public OtherCarSensor other_car_sensor;
4  public ConeSensor cone_sensor;
```

9

5.2.10 Sensor

10

5.2.11 ProcessingState

11

5.2.12 Task

Listing 12: Tasks know how to preform their actions

```
1  // says what action should be conducted
        upon task completion
2  public enum Action {
```

```
1  public class StopSignSensor extends Sensor {
2    public ArrayList<Sign> signs;
3    public ArrayList<Double> distances;
4
5    public StopSignSensor(CarPanelState state) {
6      super(state);
7    }
8
9    @Override
10   protected void compute() {
11     this.signs = new ArrayList<Sign>();
12     this.distances = new ArrayList<Double>();
13
14     for (Sign sign : this.car_panel_state.signs) {
15       if (sign.type == SignType.STOP && is_within_range(this.car_panel_state.main_car,
             sign)) {
16         // if within range of sensor
17         signs.add(sign);
18
19         // add in parallel the distance to the sign
20         distances.add(compute_distance(this.car_panel_state.main_car, sign));
21       }
22     }
23   }
24 }
```

```
3    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;
4  }
5
6  public void preform_action();
```

12 Based on their action, tasks will choose to do different things in the preform_action method. Whenever a task reads a sensor, it will start any appropriate aperiodic tasks.

5.2.13   Scheduler

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

```
1  /**
2   * returns the task set in scheduled
       order. null if not schedulable.
3   */
4  public ArrayList<List<Task>>
       schedule(List<Task> tasks, int
       n_processors) {
5    ArrayList<Task> array_list_tasks = new
         ArrayList<Task>(tasks);
6
```

```
7    return
         this.scheduling_algorithm.schedule(
8      array_list_tasks, n_processors);
9  }
```

13

5.2.14   SchedulingAlgorithm

Listing 14: Scheduling Algorithm interface

```
1  public interface SchedulingAlgorithm {
2    /**
3     * returns a schedule of all tasks for n
         processors. Each processor has a
4     * different list in the task set.
5     *
6     * @param tasks
7     * @return
8     */
9    public List<List<Task>>
         schedule(List<Task> tasks, int
         n_processors);
10 }
```

14 any number of these might be implemented

```
1  private void update_processors(long time) {
2    // preform updates in an incremental fashion
3    for (int i = 0; i < time; i++) {
4      // if a task has completed, run the scheduler at the point it
5      // completes.
6      // remember the only tasks who's computation time is decreasing are
7      // the ones in the processors.
8      for (Processor p : processors) {
9        // first, decrement the computation time of the thing in this
10       // processor
11       p.task.computation_time_remaining--;
12
13       if (p.task.computation_time_remaining <= 0) {
14         // periodic tasks need to be added back
15         if (p.task.nature == Task.Nature.PERIODIC) {
16           // add the same task back in if it is periodic
17           this.scheduler_task_queue.add(new Task(p.task.computation_time_origional,
                  p.task.period,
18             p.task.deadline, p.task.nature, p.task.action, p.task.processing_controller,
19             p.task.car_panel_controller, p.task.set_point));
20         }
21
22         // preform the complete action of the task when it is
23         // finished
24         p.task.preform_action();
25
26         // if the task has finished, get the next thing out of the
27         // queue
28         // set the current processor task to the smallest thing in
29         // the queue
30         if (p.task_queue.isEmpty() == false) {
31           // get the next task in the queuue
32           p.task = p.task_queue.get(0);
33
34           // remove this task from the queue
35           p.task_queue.remove(0);
36         } else {
37           // insert a dummy task
38           p.task = new Task(1, 0, 0, Nature.APERIODIC, Action.NONE, null, null, 0);
39         }
40       }
41     }
42   }
43 }
```

5.2.15   Processor

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

5.2.15

5.2.16   ProcessorPanel

```
1  protected void paintComponent(Graphics g) {
2    super.paintComponent(g);
3
4    drawSchedulerQueueTasks(g);
5    drawTaskTables(g);
6    drawLabels(g);
7    draw_signs(g);
8  }
```

5.2.16

5.3   Implemented Scheduling Algorithms

5.3.1   EDF

5.3.2   TODO

5.4   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.