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.

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 Introduction

3 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.

4 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.

4.1 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

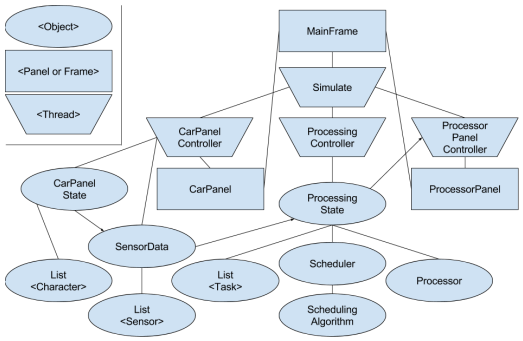

Figure 1: Program Layout

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.

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.

4.2 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.

4.2.1 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.

4.2.2 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.̈

4.2.3 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

4.2.4 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

4.2.5 ProcessorPanelController

Listing 5: Processing Panel Update Procedure

```
while (true) {
```

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();
```

```
  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

4.2.6 CarPanel

Listing 6: Ask Signs to Draw Themselves

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

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

4.2.7 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;
```

7

4.2.8 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);
  }
}
```

8

4.2.9 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;
```

9

4.2.10 Sensor

10

4.2.11 ProcessingState

11

4.2.12 Task

Listing 12: Tasks know how to preform their actions

```
// 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();
```

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.

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));
      }
    }
  }
}
```

4.2.13 Scheduler

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

```
/**
 * 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);
}
```

13

4.2.14 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);
}
```

14 any number of these might be implemented

4.2.15 Processor

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

4.2.15

4.2.16 ProcessorPanel

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

```

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++) {
    // 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);
        }
      }
    }
  }
}
```

```
4   drawSchedulerQueueTasks(g);
5   drawTaskTables(g);
6   drawLabels(g);
7   draw_signs(g);
8 }
```

4.2.16

4.3 Implemented Scheduling Algorithms

4.3.1 EDF

4.3.2 RMS

4.3.3 TODO

4.4 Outputs

5 Conclusion

5.1 Synopsis

5.2 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.

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 $50hz$.

5.3 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.