Report Simulation 2

Lukas Walbroel
Aklima Zaman
Shariar Robbani

March 27, 2017

Contents

1 Environment

1.1 Introduction

This chapter describes the parsing, generation and code of the environment-subproject. It provides the ideas behind the implementation and describes methods that were tried and discarded. In addition see the java-doc comments and the corresponding code. The project will be presented in logical parts following the logical order in which components are called. Starting with the parsing of an OpenStreetMap (OSM), followed by the necessary conversions, the generation of heights and street signs. After that we will introduce the geometric operations which are used for the computations and how pedestrians, weather and a collision detection are integrated. Finally we will introduce the usage of the environment via the central interface.

1.2 Parsing

The Environment is currently based on OpenStreetMap data. Currently every map exported at `https://www.openstreetmap.org` which is in the .osm data format can be used by the environment. Methods used for parsing are specified in IParser. To enable other data-sources add a new implementation of this interface. The result of the operations described in this chapter is an object of type VisualisationEnvironmentContainer which is a container that contains all data necessary for visualising the environment.

1.2.1 Import

Dataformat: Objects in OSM are usually represented using a list of nodes. The lines connecting subsequent nodes define the shape of such an object. Those shapes can be closed for buildings and can be open for streets. We used the same approach for streets and buildings. The class diagram is shown in figure 1.1.

The central class for the import of OSM-data is Parser2D which is an implementation of IParser. The data is imported using the osm4j-Framework.[1] This framework was chosen because it had the best documentation and has some built-in functions and objects which are supposed to make the usage of OSM-data easier. Nontheless we encountered some problems which made it necessary to implement some geometric computations by ourselves.
Currently only the import of a data file is implemented but osm4j supports other import-Methods like the overpass-API [2], too. To enable import-Methods like that replace the

[1] `http://jaryard.com/projects/osm4j`
[2] `http://wiki.openstreetmap.org/wiki/Overpass_API`

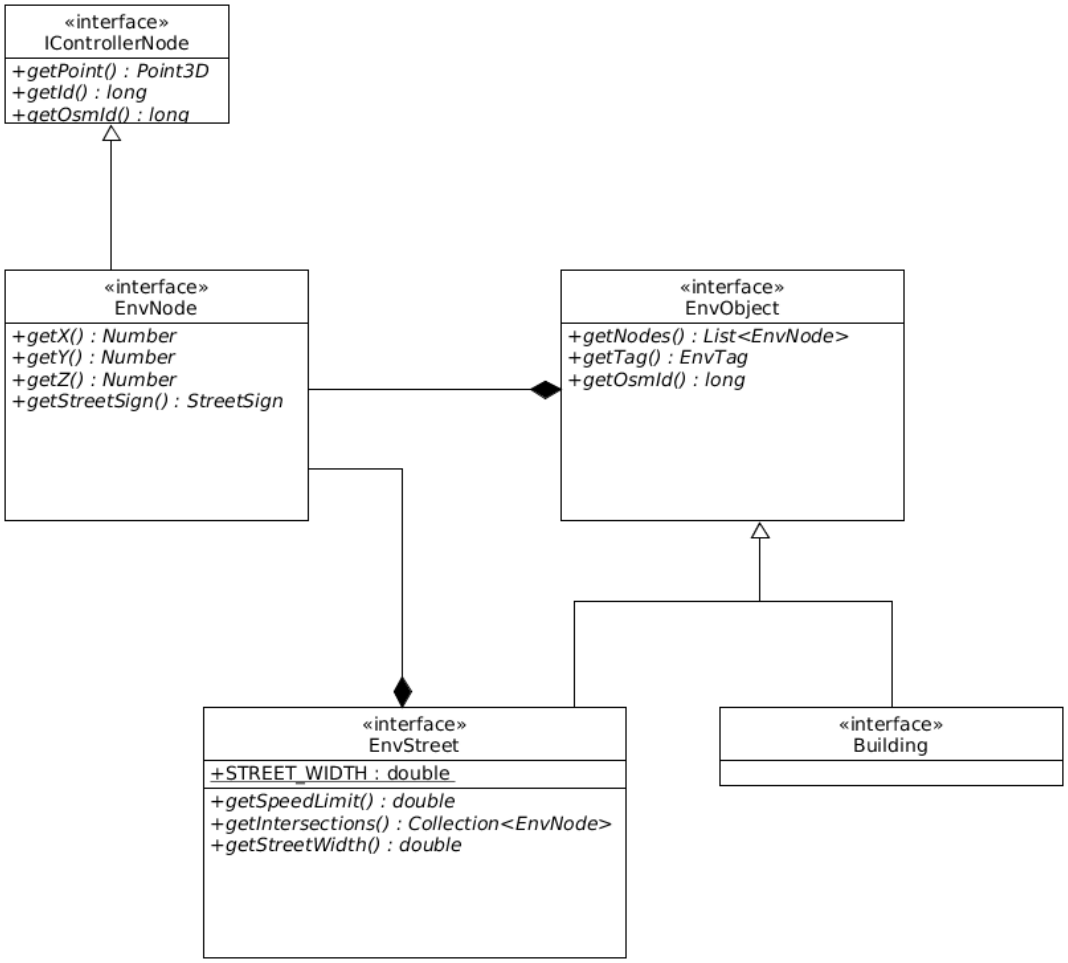

Figure 1.1: Interface-Hierarchy for environment objects

implementation of OsmReader in line 72.
After the data is imported intersections are computed and mapped to their respective streets. As described above the data uses longitude and latitude as coordinate system. In order to enable some kind of distance measurement those coordinates had to be transformed into a kilometric unit. The interface for such a converter is MetricConverter which converts the coordinates passed in the methods. The current implementation ApproximateConverter uses the method described in this post `http://stackoverflow.com/a/1253545/2451431` and converts all length units to meters. To get an idea of the precision of this method see the according test. In this version of the environment only the street running is used out of OSM, since buildings had to be discarded. For details on that see chapter 1.3.1.

1.2.2 Generation of additional data

The following data is generated in the Parser because it depends heavily on the data source. For example there might be a data source which already contains height information or a data source that contains street signs.

Heights

Since OSM delivers two-dimensional data per default, heights had to be generated. A requirement was to be able to assign a height to each point of the environment and not only those on a street. There are currently 3 possible modes to generate heights.

1. Set all heights to 0.
2. Generate heights with a fixed slope of 2 percent.
3. Generate heights randomly.

To implement a new method to generate heights implement the interface HeightGenerator in the geometry.height package. All three modes described above are given by an implementation of this interface. The first case is rather trivial, while the second one is a variant of the third one. Thus we only describe the third mode.
The random height generation is realised as follows:

1. Compute the midpoint of the environment.
2. Assign a random height to this point.
3. Put concentric circles around the midpoint until the bounds of the environment are reached.
4. The circle next to the midpoint is assigned a random slope and the height of the midpoint.
5. All other circles get a random slope and the height is calculated by using the slope and height of the next inner circle.

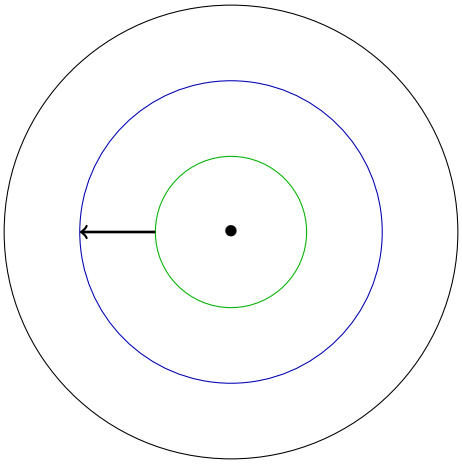

Figure 1.2: Example for the height generation

Example: In figure 1.2 the green circle has the same height h_0 as the midpoint. Assume the green circle is assigned a random slope s_0 of 0.1, and the distance between the blue and green circle (black arrow) is 20 meters. A slope of 0.1 corresponds to a height difference of $0.1 \cdot 20 = 2$ meters. This means the blue circle is assigned height $h_1 = h_0 + 2$ and a random slope s_1. Based on h_1 and s_1 the height of the black circle is determined and so on.
The slopes are currently generated by using a normal distribution with mean 0 and variance 0.1. By choosing that the probability of getting a slope which is higher than 30 percent is 0.27 percent which reflects the real world where the maximum slope of a street is 35 percent. Since we didn't want to generate too bumpy roads the choice of the slope sign is determined by taking the sign of the last slope into account. Thus if a positive slope was generated the probability to generate a positive slope for the next circle is higher than generating a negative slope.

After that heights can be generated for each point in the environment as follows:

- If a point lies between the midpoint and the first circle, it is assigned the height of the midpoint.
- If a point lies on a circle, it is assigned the according circle-height.
- If a point lies between two circles the height is generated by a linear interpolation between those two points or in other words it is defined by the distance to the inner circle and the according slope.

This method is used to generate the heights for the streets and the rest of the environment.

Street signs

Right now street signs are generated randomly for intersections. Currently the following signs are implemented:

- Egg sign (*german: Vorfahrt*)
- Stop sign
- Priority sign (*german: Vorfahrt gewähren*)
- Intersection sign (*german: Kreuzung mit Vorfahrt von rechts*)
- Traffic lights

A street sign is specified by the Interface StreetSign. Every StreetSign has a state and a type for which an enum-Type exists. Also they need an ID which is necessary for the state-updates. There are two implementations of the interface: StreetSignImpl and TrafficLight. Normal signs should be realised by StreetSignImpl and traffic lights by TrafficLight. For a normal sign the state will always equal the type since they usually don't change their state. Traffic lights on the other hand change their state over the simulation run so their state is one of the following:

- TRAFFIC_LIGHT_RED
- TRAFFIC_LIGHT_YELLOW
- TRAFFIC_LIGHT_GREEN
- TRAFFIC_LIGHT_RED_YELLOW

Street signs are generated as follows:

1. Map intersections to streets.
2. For each intersection determine randomly an intersection type.
3. Place the signs on that intersection according to the intersection type.

There are three intersection types.

Normal intersection: Each street containing an intersection like this has intersection signs at the corresponding nodes.

Egg intersection: On an intersection like this one road gets an egg sign while other streets get a stop or priority sign.

Traffic Light intersection: A street that contains an intersection like that contains a traffic light at the corresponding node. It is possible that a street contains two traffic lights with a different state at the same intersection. This happens if a street crosses the same intersection twice. Traffic lights that correspond to the same intersection are collected in a TrafficLightSwitcher. The TrafficLightSwitcher instance updates the state of the traffic lights over the simulation run. On every intersection there is only one street that has a green light. After a certain amount of time the green traffic light switches to yellow and finally to red. If a traffic light switches to red the next one switches to red-yellow and finally to green.

Note that a StreetSign instance usually corresponds to two actual signs. For the details on that see section 1.3.3.

1.3 Geometry

We already gave a short description of the OSM data format. osm4j comes with some classes that encapsulate geometric operations. Sadly those operations don't seem to interpolate streets. Therefore we had to evaluate different interpolation/splining methods in order to be able to make the necessary computations. A major requirement was to determine the distance of any point in the environment to a street and to test if a point lies on a street. Another requirement was to be able to compute any point on the street in order to simulate the movement of pedestrians.

1.3.1 Streets

We started by interpolating the streets. At that point coordinates already were in metric units. For the evaluation we used Catmull-Rome Splines and Linear Interpolation.

Catmull-Rome Splines: This method is known to produce smooth curves that pass all points on the spline. A property that isn't always given. While the computation of such splines is rather easy[3] it isn't trivial to compute the distance of a point to the spline. We used the approach proposed in `http://www.tinaja.com/glib/cmindist.pdf`, which suggests an iterative approach by using the derivatives of the splining-function.

Linear Interpolation: A simple method that puts a line between two points. It is very easy to compute the distances of a point to such a line. For the prove and source of the following equations see `http://mathworld.wolfram.com/Point-LineDistance3-Dimensional.html`. Suppose a line is defined by the two points x_1 and x_2. Any point on the line between x_1 and x_2 can be computed by

$$x_1 + t \cdot (x_2 - x_1) \text{ for } 0 \leq t \leq 1 \tag{1.1}$$

[3] `https://www.mvps.org/directx/articles/catmull/`

The distance of a point x_0 is given by

$$d = \sqrt{\frac{|x_1 - x_0|^2 \cdot |x_2 - x_1|^2 - [(x_1 - x_0) \cdot (x_2 - x_1)]^2}{|x_2 - x_1|^2}} \tag{1.2}$$

Another very handy equation is given by

$$t = -\frac{(x_1 - x_0) \cdot (x_2 - x_1)}{|x_2 - x_1|^2} \tag{1.3}$$

This equation computes the t of the point on the line on whose normal vector x_0 lies. This formula is very useful if we want to compute the z-Coordinate for a given x and y coordinate if we assume that z-Coordinates don't change on the normal vector.
The greatest disadvantage is that such lines aren't very smooth but since it is stated in the OSM-Documentation that nodes describe linear features we decided to try this method. [4]

Evaluation: For the evaluation we imported an example map. We then chose points in the environment manually specified by their longitude and latitude coordinates. For example some points were in the middle of a street and some points were a few meters away contained in a building beside the street. We then converted the points coordinates into metric units and then tested if the points on the street had distance 0 and those who weren't on the street had a larger distance. These evaluation had two results. The first one was that Catmull-Rome Splines and Linear Interpolation yielded the same results but Catmull-Rome Splines had a longer runtime. The second result was that both methods yielded wrong results. Points on streets and points that weren't on the street had distances to the street that were around the same magnitude.

Discussion: There are actually several possible reasons for the above described results. The first reason might be that the conversion from longitude/latitude to metric units isn't precise enough to detect such short differences. Another explanation might be that the OSM-data isn't precise enough. A fact that supports this thesis is that we were able to find nodes in an imported map that had shorter coordinates than the bounds specified in the map header. Actually OSM might not even need such a high degree of precision, since it is usually used for navigation purposes. Purposes like that need only a precision up to a few meters while an environment for a simulation might have to be precise up to a few centimeters. There were several ways to solve this problem. One possibility was to discard the whole OSM-structure and look for another data source or generate the environment randomly. We decided to give each street a certain width, so we could compute distances to the street borders and the middle of the street. In order to be able to generate conflict-free widths we had to make a restriction on the import and stoped importing buildings. In addition we discarded the Catmull-Rome Splines since there was no advantage in comparison to linear interpolation.

[4] `https://wiki.openstreetmap.org/wiki/Elements`

Implementation: For the final implementation a LinearInterpolator was used which interpolates between all nodes of a street. Street borders were constructed by using a movement of the difference vectors on the normal vectors. Those borders are only moved in two dimensions. This means if p is a point on the middle of a street with a height h. We turn right and walk on the normal vector to the right border of the street to a point p1. Then p1 has height h, too. Even if the height generation of p1 yields a different result. This assumption was made in order to determine the normal vectors easily and to make the computation of z-Coordinates for given x and y on a street easier. Another nice side-effect is that the roads aren't generated to bumpy. By choosing this type of implementation all the requirements are met.

1.3.2 Pedestrians

Another requirement was to simulate pedestrians. This was realised by adding pavements to the left and right of the borders of a street. A pavement is again realised by a LinearInterpolator object. There are five parameters needed for the computation of a pedestrian movement on a pavement that is between points p0 and p1.

1. distance: The distance the pedestrian moves during the current time frame.
2. pavement: The pavement the pedestrian walks on (either the left or the right one).
3. direction: either positive (from p0 to p1) or negative (from p1 to p0).
4. crossing: indicates if the pedestrian is crossing the street.
5. current position: the current position of the pedestrian.

A movement from a point p is computed by finding the point p' with the specified distance to p in the specified direction. If the pedestrian crosses the street he basically follows the direction of the normal vector. Currently pedestrians are associated with a LinearInterpolator. They walk on the pavement and follow it until they reach the end of the linear interpolation. At the end the pedestrian turns around and walks back. With a certain probability a pedestrian will cross the street. He then follows the direction of the normal vector until he reaches the opposite pavement and starts walking there in a random direction. At the beginning Pedestrians are spawned randomly on the pavements.

1.3.3 Street signs

Assume street a in figure 1.3.3 gets an egg sign, street b a priority sign and c a stop sign. Since b and c end at the intersection both StreetSign instances correspond to one actual sign. The priority sign is at the western end of the intersection and the stop sign on the eastern end. Since a doesn't end at the intersection the egg sign has to be placed at the northern and southern end of the intersection. Note in the current implementation nodes are compared based on their coordinates. Nontheless every node is a different instance. It is possible that there are two instances of the same coordinates in one street's node

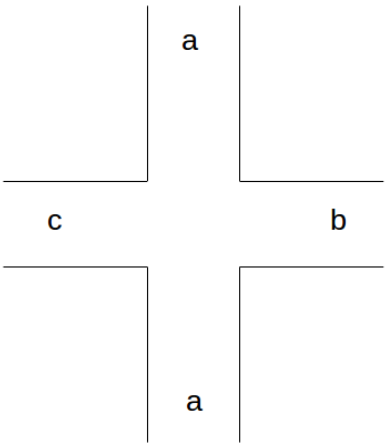

Figure 1.3: Example intersection

list. For example if a street crosses the same intersection twice. Assume a makes a turn and leads into what is in figure 1.3.3 b. By the use of different instances it becomes possible to assign the same intersection different street signs. In the described situation the signs themselves wouldn't change but b would become a.

Positioning: A StreetSign instance can have up to 2 positions. They are set as follows:

1. determine the right (left) border of the intersection
2. move a specified distance in direction of the predecessor (successor) if there is any
3. place the sign at the determined position

The distance is currently specified as the half of the street-width. As long as the the distance to the predecessor (successor) is greather or equal to the half of the street-width. If that is not the case the right (left) border of the predecessor (successor) is used as position. This might not take all possible cases into account.

1.4 Weather

Currently the environment supports rain and sunshine. There are three modes for the weather:

- Fixed weather: The weather doesn't change and keeps being the way it is initialised.

- Fixed weather changes: The weather changes randomly at fixed time intervals. For the generation of the weather an equal distribution is used
- Random: The weather changes randomly at random time steps. For the generation of the time intervals an exponential distribution is used.

The weather itself is realised by a double value with everything ≤ 0.5 being sunshine and rain otherwise.

1.5 Collision Detection

The simulation normally works with cuboids. Generally it would be sufficient to describe every cuboid as BoundingBox and test those for intersections. Since the corresponding classes in javafx and apache-commons don't support rotations we decided to test the boundary vectors vor intersection. There are methods in apache.commons which test vectors for intersection but those methods only support infinite vectors while the boundaries of the simulated cuboids are finite. Therefore the test for the intersection of two vectors was implemented for such finite vectors. Any intersection can generally be described as $\vec{x}+t\cdot(\vec{y}-\vec{x}) = \vec{v}+u\cdot(\vec{w}-\vec{v})$. This equation can be solved for t to determine the coordinates of the intersection. Let $\vec{a}, \vec{c}$ be origins of two boundary vectors and $\vec{b}, \vec{d}$ be the difference vector to their corresponding end and x_i be the i'th component of vector x. Solving the equation above for t leads to

$$t = \frac{d_2 \cdot (a_1 - c_1) - a_2 \cdot d_1 + c_2 \cdot d_1}{b_2 \cdot d_1 - b_1 \cdot d_2} \tag{1.4}$$

A collision is detected if $0 \leq t \leq 1$ holds. Obviously the equation holds not in all cases. There are some other cases to be considered.

1. $\vec{b} = \vec{d} = \begin{pmatrix} 0 \\ 0 \end{pmatrix}$: If this case holds $\vec{a}$ and $\vec{c}$ are points. A collision happens if they have distance 0.
2. Another case that has to be checked if one of the vectors is a point. We then have to solve for t in both components and check if the results are equal to each other. (If a component in the difference vector of the line is 0 this has to be considered as well)
3. The third case to be considered is if the lines move only in dimension i. Collisions can only happen if the values of the non-moving component are equal in $\vec{a}, \vec{c}$. If that is the case we have to check if a point of the first line lies between the two points of the other line and vice versa.
4. The last case to be considered is if the two difference vectors are equal or multiples of each other. For a collision either $\vec{a}$ or $\vec{c}$ has to lie on the other line thus a solution for either $\vec{a} + t \cdot \vec{b} = \vec{c}$ or $\vec{c} + t \cdot \vec{d} = \vec{a}$ has to exist with $0 \leq t \leq 1$ or $0 \leq u \leq 1$ which can be easily determined.

1.6 Usage

The environment can be acessed by using the World-Interface. The current implementation is called WorldModel. It's a singleton and can be initialised by WorldModel.init() and accessed by WorldModel.getInstance().

1.6.1 Initialisation

WorldModel.init() needs two parameters. First one is an object of type ParserSettings and the second one is of type WeatherSettings. Both are simple data containers that specify parameters needed for the initialisation. ParserSettings has two attributes which can be set by the according constructor:

- String in: This parameter specifies the path to the map-Ressource. Current maps are located in 'src/test/data'
- ParserSettings.ZCoordinates z: This enum specifies one of the three modes for height generation described in Chapter 1.2.2

WeatherSettings has three constructors. One for each weather mode described in Chapter 1.4.

- The Constructor without parameters will initialise the Weather to be completely random.
- long fixedWeatherChanges: This constructor will initialise the weather to be changed in fixed time intervalls.
- double fixedWeather: This constructor initialises the weather to never change. You can use the public constants in weather.Weather for this constructor.

1.6.2 Interface-Methods

The World-Interface provides several methods needed by other components.

```
public abstract Number getGround(Number x, Number y, Number z);
```

This method provides the correct z-Coordinate for a point in the environment. Note that x- and y- have to be the precise coordinates of the point while z has to be equal to the last known z-Value. Although this value wouldn't be needed at the moment it will become necessary if the map becomes multi-layered. Assume there is a street which contains node $(x, y, z1)$ and another street containing node $(x, y, z2)$ with $z1 \neq z2$. In order to compute the right z-Coordinate for x and y a street has to be chosen. This is done by computing the distance to this street by using the last known z-Coordinate and then choosing the street with the shortest distance. After that the nearest point on the street is computed for x and y and the z-Coordinate of that point is returned. If the point isn't part of a street the z-Coordinate of the height generator is returned.

```
public abstract Number getGroundForNonStreet(Number x, Number y);
```

This method returns the z-Coordinate returned by the heightGenerator even if the point lies on a street. Therefore no additional z-Coordinate is needed as parameter.

```
public abstract Number getDistanceToMiddleOfStreet(PhysicalObject o);
```

This method returns the distance of the PhysicalObject to the middle of the street it is contained in.

```
public abstract Number getDistanceToLeftStreetBorder(PhysicalObject o);
```

This method returns the distance of the PhysicalObject to the left border of the street in the direction of travel. Note that in the current implementation this method returns the 2D-distance thus a height correction didn't have to be done in the sensors.

```
public abstract Number getDistanceToRightStreetBorder(PhysicalObject o);
```

This method works analogously to getDistanceToLeftStreetBorder.

```
public abstract VisualisationEnvironmentContainer getContainer()
        throws Exception;
```

provides a representation of the environment for visualisation.

```
public abstract boolean isItRaining();
```

returns true iff it is raining in the simulation.

```
public abstract double getWeather();
```

returns a double representation of the weather.

```
    public abstract ControllerContainer getControllerMap();
```

returns a representation of the environment for the controller-module. Primarily meant for navigation.

```
public abstract PedestrianContainer getPedestrianContainer();
```

returns a container for all pedestrians in the environment

```
public abstract Point3D spawnOnStreet(Number x, Number y, Number z,
                                      boolean rightLane);
```

returns a Point which resides on the middle of the specified lane on the street with the shortest distance to the coordinates in the street.

```
public abstract Point3D spawnNotOnStreet(Number x, Number y, Number z);
```

returns a point which has x- and y-Coordinate as specified or null if x and y are on a street.

```
public abstract List<Long> getChangedTrafficSignals();
```

returns a list of the traffic light id's which changed their state in the last timestep.

2 Sensors

2.1 Introduction

Testing autonomous vehicles in real world must be a risky and costly task. As a result the necessity of virtual simulation has increased which can defiantly reduce the cost and increase flexibility and safety of autonomous vehicle testing. In real world autonomous vehicle can sense the environment via sensors. Sensors are in the real world could not produce actual value because the produced value contains various kinds of noise. On the other hand, virtually simulated vehicle have no chance to produced the sensing values with noise. So, we could never test the real world situation of the autonomous vehicle as long as we do not have any noise in our sensing values. To make the virtual simulation realistic, we need to add some noise modes to the virtual sensors, which are similar to noise modes of the real world sensors.

2.2 Architecture

2.2.1 Sensor Interface

All sensors in this project should implement the specification of this sensor interface. Additionally, This is the public API to use a sensor Object. Diagram 2.1 describes the architecture of the sensor interface.

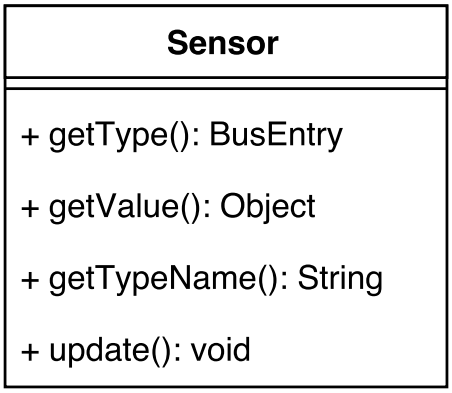

Figure 2.1: Sensor Architecture

- **getType():** Returns a BusEntry. To pass the value to controller a databus is used where all values are kept in key-value pair manner. For data bus, keys are defined

in BusEntry and for each sensor there should be a specific unique BusEntry. Main purpose of this method is to save the sensor value in the DataBus with getType() as a key and getValue() as value.

- **getValue():** Returns the calculated value of the sensor. Before calling getValue() one should call update() method which calculates the value otherwise value would be null. The return type is kept as Object type because different kind of sensors produce different type of values. For example, camera sensor returns an Image object where velocity sensor returns Double object. The Object type of a sensor value can be get as String by calling getTypeName() method.

- **getTypeName():** It returns the object type of a sensor value. The type name is a String which contains full path of a Java Class like <package-name.Class-Name>. For example, for velocity sensor this method returns java.lang.Double on the other hand for camera sensor the type name will be java.awt.Image.

- **update():** It does not return anything but it contains the main calculation of the sensor. So, to produce sensor value on should call this method.

2.2.2 Class Diagram

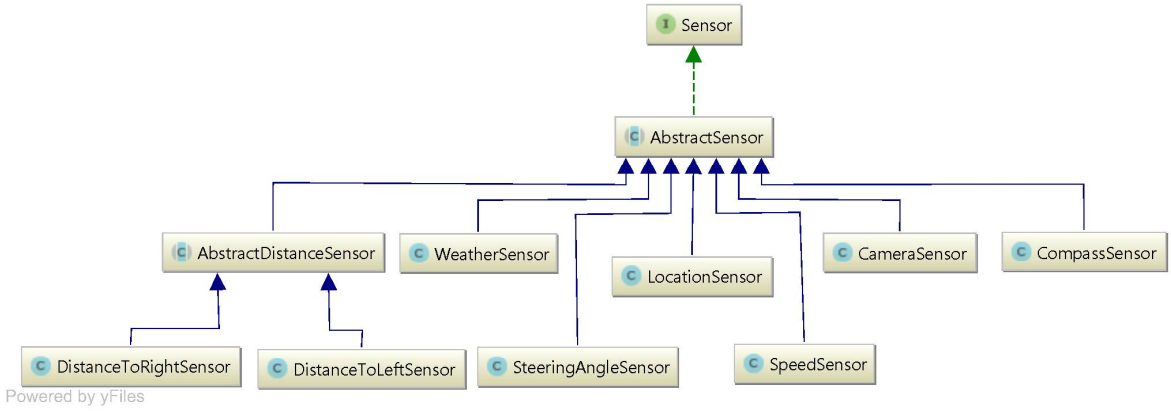

Figure 2.2: Abstract Sensor

2.2.3 Abstract Sensor

It is an abstract class implementing the sensor interface. All sensors should extends this class. It has only one constructor. The constructor has one parameter which takes a Physicalvcehicle object as argument. This class also contains an abstract method called

calculateValue(). The sensors who extends this update() class should have to implement calculateValue() method. This calculateValue() will contains the main calculation of the sensor. The update() method is implemented in this AbstractSensor class. In the update() method the calculateValue() is called. By this architecture one could put some checks in the update() method before calling calculateValue(). For example if someone creates a sensor Object with null Physicalvcehicle object, we can check before calling update if the Physicalvcehicle object is null, and if it is null then calculateValue() method should not be called.

2.3 Implemented Sensors

All of the implemented sensors extends Abstract Sensor and each of them have one constructor which takes a Physicalvcehicle object.

2.3.1 SpeedSensor:

- **Return value type:** java.lang.Double
- **Data Source:** Speed sensor extracts velocity value from physicalvehicle object which is a vector. After that, Norm [1] function is applied in Velocity vector and converted to double value.
- **Noise Model:** The noise model for velocity assumed to be a Gaussian Distribution. With the standard deviation of 0.3, we generate a random value sampled from normal distribution and this value is chosen as calculated sensor value.

2.3.2 SteeringAngleSensor:

- **Return value type:** java.lang.Double
- **Data Source:** Extracts steering angle value from simulated vehicle object of physicalvehicle object. The value of steering angle we get from simulated vehicle is double.
- **Noise Model:** The noise model for SteeringAngle is same as velocity.

2.3.3 LocationSensor:

- **Return value type:** org.apache.commons.math3.linear.RealVector
- **Data Source:** Extracts position value from physicalvehicle object which is a vector.
- **Noise Model:** In real world, GPS sensor receives signal from satellite and based on the position of the satellite it calculates the vehicle position. So the accuracy of signal is highly depends on the weather and how satellite can see the GPS Module.

For example, if one vehicle is in the under pass it is quite impossible to calculate the position of the vehicle. It is also true for bad weather. If it is snowing or storming, the value would be more inaccurate. For calculating the position of the vehicle we use Gaussian Distribution same as SpeedSensor but the standard deviation increase or decrease according to the weather condition.

2.3.4 CameraSensor

- **Return value type:** java.util.Optionalof(java.awt.Image)
- **Data Source:** Extracts Image from simulated vehicle object of physicalvehicle object. The image is in stereo mode[2.3]. First of all the stereo image is divided into two parts, one is the left image[2.4] an another is the right image. The default getValue() method will always returns the left image.
- **Noise Model:** The noise model of CameraSensor depends on motion of the vehicle. If vehicle is in very high speed, the captured image will have motion noise[2.6]. Another noise could be perspective distortion of image[2.5]. Currently those noise are not yet implemented because of performance. Those filters are applied in every pixel of the image and the processing becomes very slow. But the test cases are provided.

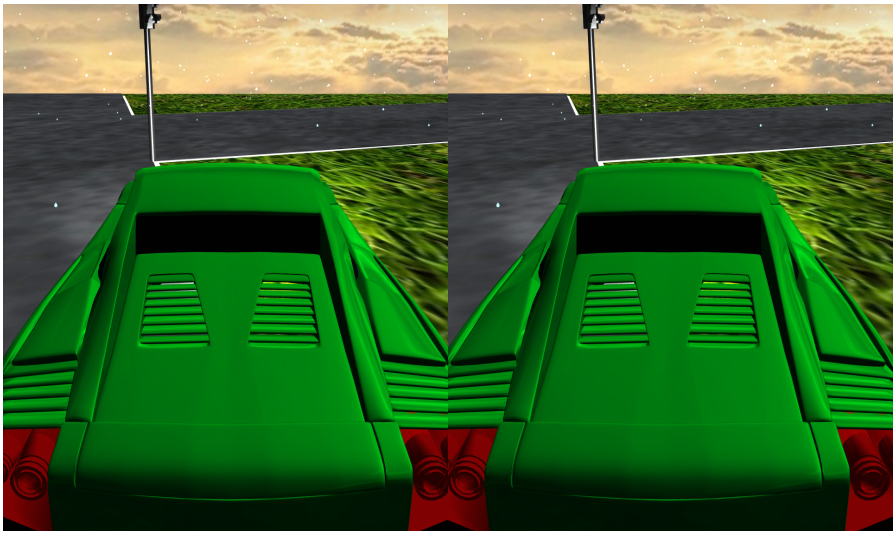

Figure 2.3: Extracted Stereo Image

Figure 2.5: Perspective Distortion

Figure 2.6: Motion Blur Distortion

Figure 2.4: Left image after Cropping

3 Integration of Visualization and Simulator

3.1 Introduction

In this project simulator, simulation objects, environment are maven module. The visualization module is using simulator and all other modules as external library.

3.2 Visualization Module Description

Visualization contains two parts, one is server side code (spring mvc) and other is javascript client. Java server side code (SimulationLoopNotifiableController.java) sets up the simulator, instantiate the simulation objects, also register them into the simulator and then it starts the simulation. This controller then store the states of all simulated objects in a list. Meanwhile, client asks for the next frame with current screen-sort and then controller sends the state of next frame. When the controller receives screen-sort it saves it with the car id in a static hash map. Before each simulation loop the screen sort is set to the physical vehicle.

3.3 Communication Procedure

When the client loads the page, the server establishes web socket connection which is used after the start of a simulation to push simulated data (mainly car positions and speeds) to client for rendering.

However, not all the information is exchanged through the web sockets. Starting, stopping and map loading are done through plain HTTP requests because these operations are typically done once only.

3.4 App Properties

There is a file app.properties where some properties are defined. If someone wants to disable the screensort finctionality it is possible by just changing enablescreensort property. There are some other property available.

3.5 Process

The simulation process works in the following way:

1. The client requests the main page (HTTP GET)

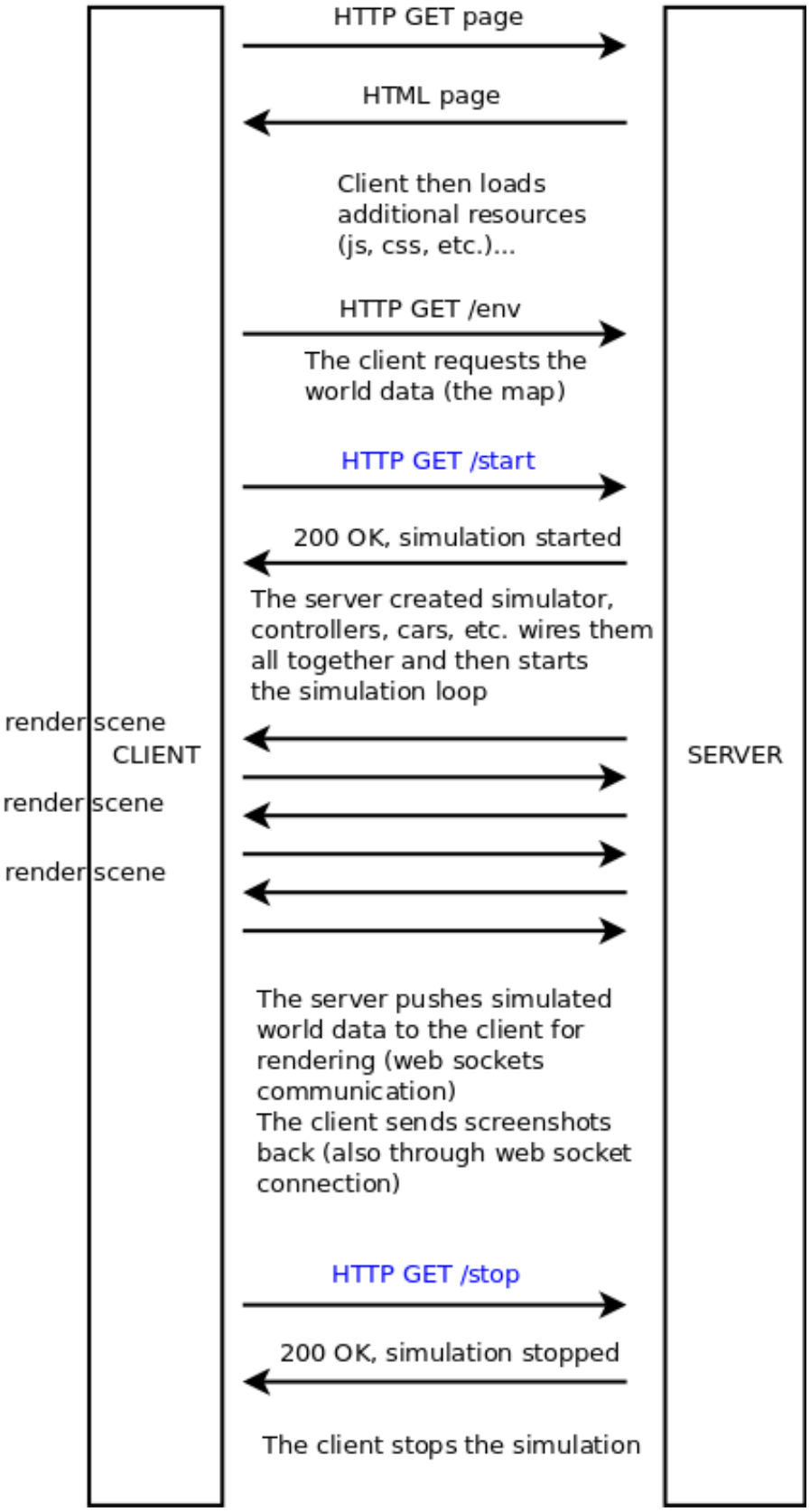

Figure 3.1: Process Diagram

2. The server returns the page
3. The client then requests additional resources (js scripts, css styles etc.)
4. Then the client also requests the world information i.e. the map (HTTP GET /env)
5. At this moment, the web socket connection is also established but is not yet used
6. The simulation is fired with the HTTP GET /start request
7. Upon this request the server starts the simulation
8. During the simulation, the server pushed the simulated world data (cars positions, cars speeds etc.) to the client for rendering through the established earlier web socket connection
9. The client renders the scene and sends screenshots back to the server (also through the web socket connection)
10. In the end, the client sends HTTP GET /stop to stop the simulation
11. The server upon this request stops the simulation.

Bibliography

[1] Norm https://books.google.com.au/books?id=hdkytqtBcyQC