CHAPTER 34

OTFVis: MATSim's Open-Source Visualizer

David Strippgen

34.1 Basic Information

Entry point to documentation:

`http://matsim.org/extensions` → otfvis

Invoking the module:

`http://matsim.org/javadoc` → otfvis → `OTFVis` class, `RunOTFVis` class

Selected publications:

Strippgen (2009)

34.2 Introduction

For most MATSim users, Via's (Chapter 33) free branch will be a good solution for their visualization needs. However, if project demand reaches beyond the given (and fixed) abilities of the Via free version, there is another—though not as stylish—option for MATSim output visualization, the OTFVis.

The short term for "On the Fly Visualizer", OTFVis was designed to support actual visualization of live simulation runs with MATSim. Therefore, one purpose of the OTFVis is the debugging of MATSim (input) data. Nonetheless, playing prerecorded movie (MVI (An OTFVis Movie File, not to be confused with the "Musical Video Interactive" file usually abbreviated mvi)) files created from MATSim events is another way to use OTFVis. Generally speaking, OTFVis serves as an open-source counterpart to the possibilities Via gives the MATSim community. The OTFVis is written in Java and available as source code to extend for different MATSim projects' special needs. Hence, it is possible and desirable to actually extend the OTFVis functionality, incorporating the user's own data sets and visualizations.

How to cite this book chapter:

Strippgen, D. 2016. OTFVis: MATSim's Open-Source Visualizer. In: Horni, A, Nagel, K and Axhausen, K W. (eds.) *The Multi-Agent Transport Simulation MATSim*, Pp. 225–234. London: Ubiquity Press. DOI: http://dx.doi.org/10.5334/baw.34.

34.3 Using OTFVis

In this chapter, we show how to achieve simple things, like creating MVI-files from MATSim run events, how to play these MVI-files and how to use a MATSim config file to view/play an actual simulation with all data (e.g., agents' plans) attached. With the latter, it is also possible to examine the data "on the fly" by sending queries into the mobsim and visualizing the results.

34.3.1 MVI Files

MVI files can be generated through the OTFVis. Under the hood, these files consist of a few binary dumps of OTFVis data packed into a zip-file. This binary data is created by Java's own serialization capabilities. Unfortunately, this setup is not very change-resistant, making it advisable to regard MVI files as temporary cached versions of your event files. These MVI files can be re-created at any time from the event files. Still, as converting one into the other is a time-prone process, the MVI files are a handy tool for temporary storage and fast loading of your visualizations.

34.3.2 Starting OTFVis

OTFVis is a MATSim contribution. There is no actual stable release of the OTFVis package; so, to acquire a working version, a "nightly build" needs to be downloaded as shown in Section 44.3.6. There, one finds the latest `otfvis-version-SNAPSHOT-build.zip` file available for download. Unzip it to the place where the `matsim.jar` already resides; do not forget to extract the `libs`-directories found in the respective zip files.

OTFVis demands substantial RAM (depending on your simulation size/MVI file); to successfully launch the visualizer, a command line like

```
java -Xmx500m -cp MATSim-XXX.jar:otfvis/otfvis-XXX.jar
   org.MATSim.contrib.otfvis.OTFVis
```

(exchange ";" with ":" depending on the used OS (Operating System)) is a good starting point. This will open the dialog window shown in Figure 34.1, asking for one choice from four possible usages of OTFVis; these will be explained in the next section.

34.3.3 Use Cases of OTFVis

With the open dialog appearing after starting the vanilla `OTVFis` class, the following options appear, as shown in Figure 34.1:

1. opening a prerecorded MVI file,
2. opening a network file (for inspection),
3. opening a live run of a MATSim config file (rather memory intensive) or
4. converting an event file (plus a given network file) to a movie (MVI) file.

Each tab stands for an individual usage. To start a visualization, one chooses the appropriate tab, fills in the necessary data and finally proceeds by pressing the `Load...` button located in the bottom left corner of the window.

The next sections provide an overview of different ways to use OTFVis.

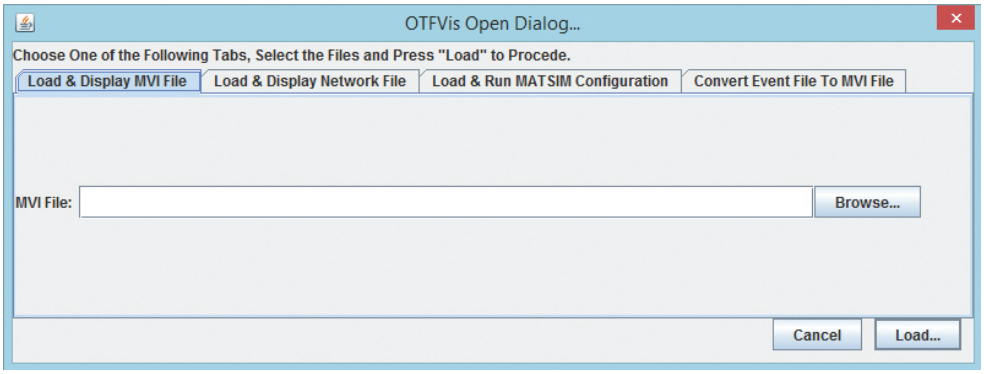

Figure 34.1: OTFVis Start Dialog.

34.3.3.1 Converting Event Files

Though the first option tab is the most used choice for OTFVis, the fourth, and last, option tab is a good starting point for exploring the visualizer; after having successfully run a MATSim simulation, there will typically be some event files at one's disposal. With any of these event files and a given (matching) network file, a MVI file can be created. Four items: event, network and movie file names, as well as a time period, must be specified for this tab to execute. The last parameter is a time period, after which a new sample of the mobsim's state is taken. This MVI-generation process might be time consuming. For smaller projects, it might be an option to display the outcome in the visualizer right away (by checking the box `Open mvi afterwards`). If the choice is to just convert the events to a MVI file, this can be opened with the first option tab of the visualizer's start dialog at any time.

From the shell, this process can be started by giving the event file, network file and, optionally, the conversion period as input parameters.

34.3.3.2 Network File Loading

The second option tab offers the opportunity to examine a network file (e.g., for errors). It will show a rendering of the given network and also, if so chosen in the preferences, the associated network link IDs for each link. This option might be helpful for debugging a freshly converted network, or inspecting specific regions and connections. Loading and interacting with a network file should be very fast.

The network file can also be given as the sole parameter to OTFVis with the shell command.

34.3.3.3 Running a MATSim Configuration

The third, and most advanced, option for running OTFVis is an actual, live running mobsim, visualized in real time (actually much faster than real time; who has all day to watch tiny cars drive around?). This option includes the possibility of exploring the data set and issuing queries into the executing mobsim. These queries can display an agent's day plan, show all links driven by agent's crossing a particular link of interest, search for a particular link or node by ID, or answer any user-defined queries. We will see later in this chapter how to program a user's own queries, but for the rest of this section we will detail OTFVis "offline" behavior.

It is also feasible to input the config file as a single parameter to OTFVis by starting it from the shell. OTFVis will make an educated guess whether the input is a config or a network file.

34.3.3.4 Loading & Displaying an MVI File

If the first and default option tab is chosen, a MVI file is selected and shown as detailed in next section 34.3.4. This is the most common use case for OTFVis; the same results can be achieved by starting OTFVis from the shell with an MVI file as an argument.

34.3.4 Viewing an MVI File

An example is illustrated in Figure 34.2. On the top left of the application, one finds buttons for controlling the file playback. A short summary of the functionality is given in Table 34.1.

This buttonbar is followed by a text field where the desired time can be written for an instant jump. In an MVI file, one can jump forward and backward in time, whereas in the live simulation case, going back in time is omitted.

Another way of iterating through the animation is to grab the time slider at the bottom of the application and drag it. Opening and closing bracket symbols are located on the left side of the slider; by clicking them, one can set the start, or end, time of a time loop to the actual time step given, making it possible to restrict playback to a certain space of time.

34.3.5 General Interaction with the Main Screen

Regardless which option for loading data was chosen, interaction with the main display area is the same.

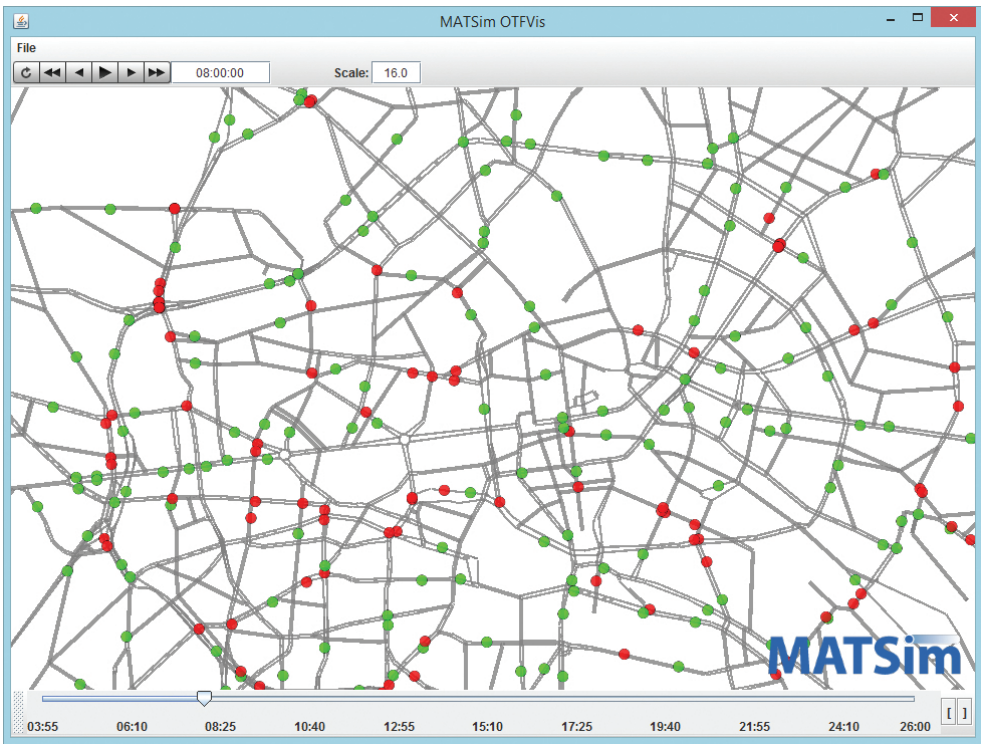

Figure 34.2: Displaying an MVI file.

Icon	Function
	Reset - set time to the start time
	Large step back
	Small step back
	Play
	Pause
	Small step forward
	Large step forward

Table 34.1: OTFVis Buttonbar.

Right button drag: Extend a rectangle for zooming into the view. Releasing the button will execute a zoom, so the chosen rectangle will best fit the screen.
Middle-Mouse-drag: Pan (translate) the screen.
Right-Mouse-Click: Show a context menu (for now only with the option to save the view settings).

34.3.6 User Interaction in the Live Mobsim

When started as a live simulation, OTFVis will look different than Figure 34.3. First, the controls of the simulation's view flow are a restricted subset of those used in MVI playback. There is no way to reset or rewind the simulation. One can still take small or large steps forward. A new option

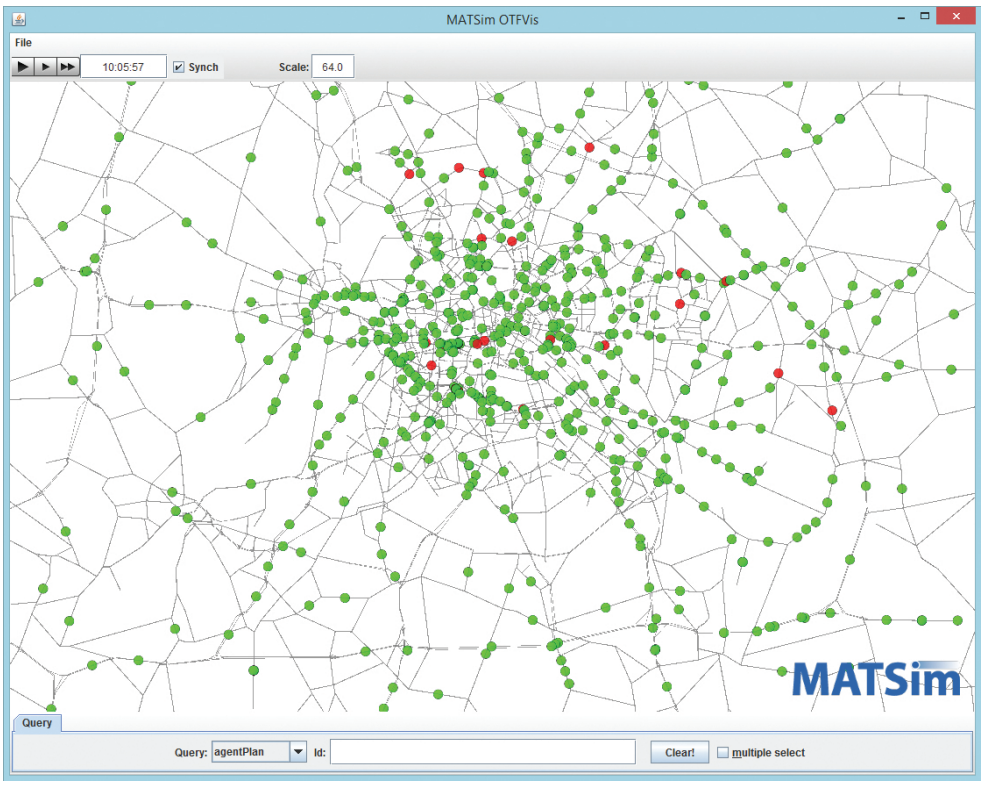

Figure 34.3: Live mode.

is given by the `synch` checkbox, which determines whether the mobsim will stop for each frame the OTFVis renders, or run independently. Usually the `un-synched` version will proceed faster, as the OTFVis output is restricted to a default of about 30 frames/updates per second and a small mobsim's simulation speed will be a magnitude higher. The time-consuming generation of visualization data will also only be necessary for a small fraction of the simulation. Length of OTFVis pauses between frames can be configured in the preferences dialog.

Apart from the reduced control set, there is another UI element new to this OTFVis option. At the bottom of the screen, the scrubbar/time line element is replaced by a "query" bar. It is possible to code "queries" into the mobsim, answering questions about its inner state. As the simulation is actually happening, all information necessary to run it is available for output. This is a clear superset of information available in the event files and in the MVI files. This rich information infrastructure can be queried and visualized in many ways. In the next session, a query example is given.

34.3.7 Running a Query in OTFVis Real Time Data

From the dropdown box, one can choose the different query types. Often, additional input is necessary, either in the text field next to it or, more often, by clicking into the network. To give an example with `agent query` selected, a click onto any agent's symbol will give a visualization of this particular agent's day plan. This is shown in Figure 34.4. There are other pre-defined queries. These queries are rather project-oriented, so defining own queries will probably be necessary to make best use of this option. In the second part of this chapter, we will look into defining own queries.

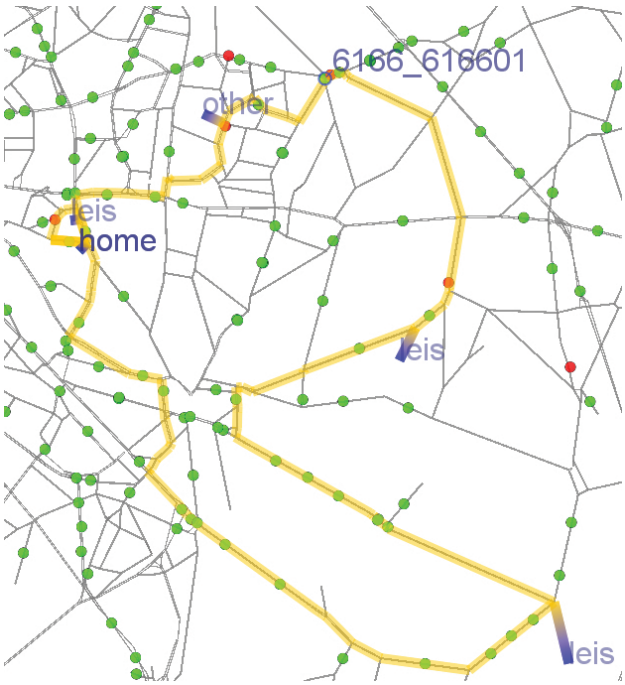

Figure 34.4: Queries.

34.4 Extending OTFVis

Because it is open source, the OTFVis is a good starting point for customizing mobsim run visualizations. OTFVis has been written in Java, but depends heavily on the JOGL (Java OpenGL) Java library. JOGL is a very thin layer within the OS hardware driver, meaning it will have OS-specific, native dependencies. These should be attended to by the maven-dependency management, but should still be kept in mind when developing for OTFVis. The displaying parts of OTFVis are based on OpenGL (Open Graphics Library). Therefore, it will be necessary to understand OpenGL to create new ways of displaying data. In the following sections, we examine how data is computed inside the OTFVis and how this can be extended.

34.4.1 Design Principles of OTFVis

The overall goal of OTFVis design was to have an easy-to-extend, fast visualizer capable of handling huge amounts of data. The specific design goals for the visualizer were:

- abstract data source (data collection) from data display (visualization),
- easy extension with own data types,
- capability for local simulation run on desktop computer,
- reduction of sent data to a minimum,
- visualization that connects to running simulation (on-the-fly),
- minimally-invasive format for existing MATSim code,
- enough speed for large scenarios,
- visualization that reads from post-mortem dump (MVI file), and
- use of hardware support for drawing.

MATSim runs can easily engage millions of agents traveling a network. To make a visualization of these large data sets feasible, two measures have been taken. A quad tree structure was implemented to ensure that only the smallest set of data necessary to display the visible sector of the network is transferred. The quad tree is a simple data structure to aggregate spatial data and retrieve parts of it efficiently for real time visualization. Apart from data structures, hardware is also used to speed up displaying the simulation. OpenGL is a platform-independent API for interfacing graphics hardware, specifically the 3D acceleration chips implemented in every contemporary computer. With the aid of 3D graphics hardware, millions of agents can be displayed in real time. Other measures were taken to segregate data extraction from data visualization, like the reader/writer pairs presented in the next section.

34.4.2 Readers and Writers

OTFVis was designed to be minimally dependent on the mobsim used. Data formats applied within the mobsim should be abstracted from data used in OTFVis, meaning that any data passed to the visualizer will have to run through some stages of abstraction.

The first stage is a writer-reader pair, responsible for transferring a certain set of data to the OTFVis. The writer will understand the data format of the hosting mobsim and convert it to simple data types, like float or string values. A set of these writers, all using a joint byte buffer to aggregate the data, will be called after each mobsim step to accumulate data. This array of bytes is then sent to the visualizer, which, in the original design, could be run anywhere in your network.

For each writer, there has to be a sibling-reader class, responsible for reading back extracted data from the byte buffer. It is crucial to ensure that these pairs work synchronously. Most

Writer/Reader-pairs are implemented in the same class, since having the source-code at the same place reduces errors in the synchronization.

Apparently, it can be necessary, or at least useful, to have different ways of visualizing data on the OTFVis front-end. Thus, actual readers are not responsible for the drawing of a certain data set. A third kind of class is responsible for that, the drawer classes.

34.4.3 Visualization of the Data

The reader objects in the quad tree will generate separate drawer objects for displaying "their" information and add these to another data structure, called `SceneGraph`, which is responsible for the actual drawing onto an OpenGL canvas. Displaying data in an interactive application will make re-draws of the display necessary for a variety of reasons: displaying menus, animations, zooming, panning and other user interactions. Not all of these changes introduce new data from the mobsim. Zooming into the network will not imply reading data from the mobsim; panning the view most certainly will. When no new data is needed, the scene graph is capable to handle all operations, no reader/writer class will be accessed and displaying is solely done with existing drawers. On the other hand, if new data is demanded, the scene graph will be "invalidated" (a term lent from the OpenGL community); thus, the graph will be dismissed and all relevant readers will be asked for new drawer objects representing the actual view. The scene graph is mainly a list of drawer objects; as an extra structuring unit, these drawers can be sent into different layers, to render them more effective.

34.4.4 Layers

To make sure that only data actually necessary for drawing the particular area visible in the viewport is sent, writers should minimize the data packets, so the quad tree can make a spatial data reduction. This seems somewhat in opposition to OpenGL or any graphics API). The API wants maximal data to be accumulated, to optimize output through the underlying hardware graphics pipeline. Think of an assembly line vs. a handcrafted item; whenever the flow of data is interrupted, the assembly line stalls and graphics performance derogates. To ease this issue, "layers" have been introduced to OTFVis. Any drawer (responsible for a bit of information) can be assigned to a layer and these layers will ultimately be summoned to draw the screen's content. It is up to the layer to optimize the execution of the drawers when necessary. For example, a network layer might store all network info from the drawer in one array, or display a list to optimize drawing of the network; (often in OpenGL, it is advisable to rather let the hardware decide what to draw. It might be faster to have all complete data residing in graphics hardware memory, rather than to transfer the reduced information set every frame). There are three layers predefined in OTFVis. The `networkLayer` contains the static street net, the `agentLayer` the actual dynamic agents and a third layer, the `miscellaneousLayer`, contains additional data.

34.4.5 Patching the Connections

In total, there are four basic elements involved in the visualization: writers, readers, drawers and layers. An additional class configures how the first two work together: `OTFConnectionManager`.

This class maps several routes for the information coming from the mobsim, building a chain of responsibility. Each data item starts at a link in our network. An `OTFDataWriter` object is responsible for extracting the desired data from the link and writing it into a `ByteBuffer`. Complementing this, an associated `OTFDataReader` is needed to retrieve data from the buffer. This item will also be responsible for adding a drawable item derived from the

class OTFGLAbstractDrawable to the scene graph representing the actual screen content. The connection between these items is made by adding entries into the OTFConnectionManager, with calls to OTFConnectionManager.connectLinkToWriter(OTFDataWriter) and OTFConnectionManager.connectLinkToWriter(OTFDataWriter, OTFDataReader), respectively.
Example (from the OTFClientLive.java):

```
conMan.connectLinkToWriter(OTFLinkAgentsHandler.Writer.class);
conMan.connectWriterToReader(OTFLinkAgentsHandler.Writer.class,

  OTFLinkAgentsHandler.class);
```

34.4.6 Sending the Data

The class OTFLinkAgentsHandler should give a good example of extracting, sending, receiving and displaying data in the OTFVis context. The method invalidate is called whenever the actual scene graph has been dismissed and needs to be rebuilt. In this case, a valid representation of the object's state should be added to the new scene graph. This also means that for drawing the actual scene, no additional reading will take place, unless there is a change in the visible data: then, this update is triggered.

34.4.7 Performance Considerations

When implementing new ways to visualize data, the following guidelines should be kept in mind.

If the data is spatially distributed over the whole network and is updated frequently, an OTFDataWriter/Reader pair should be considered. It will reduce data updating to times when the data is actually visible, not creating, transporting or drawing the data otherwise. If a fraction of the data needs to be transferred only once—because it is static over the time of the simulation—it can be sent with the writeConstData() method; otherwise using writeDynData() is advised. If the data is sparse and little information is transmitted or it has no discernible spatial cohesion, it might be simpler to just add it to the server quad tree as additional data with a call to OTFServerQuadTree.addAdditionalElement().

34.4.8 Sending Live Data

Flow of data within OTFVis is almost always a one way affair, except for one important issue: sending queries into the simulation. In case of a live simulation run, visualized with help from the OTFVisLiveClient class, queries can be sent into the simulation. Again, the methods involved in this process are threefold; queries will be realized through an object derived from the abstract class AbstractQuery. Such an object initiates several methods that will be used as callback over the lifetime of the query.

First, a new query is sent to the server and the method installQuery() is called. In this method, all relevant parts (network, population, events) of the simulation run can be accessed and data can be collected. The visualizer framework will later repeatedly call the result() method, to retrieve an OTFQueryResult object. This has to implement a draw() method, to visualize the results in the given screen context. If the result indicates isAlive(), the query() method of the AbstractQuery-derived object will be called with each frame; otherwise, only once.