Traffic 3.0 –
Introducing time into a city model

Semester Thesis

By: Florian Geldmacher
Supervised by: Michela Pedroni
                        Prof. Bertrand Meyer

Student Number: 02-911-774
1. Introduction

This document is a short description of my semester project. The project was split into two parts. In the first one I extracted a map widget out of the City3D application. This task included developing a basic TRAFFIC_3D_MAP_WIDGET as well as implementing first versions of the REPRESENTATION classes, which are container for the different 3D objects, and the corresponding FACTORY classes, which create the 3D objects. You will find the description of this task in the next chapter.

The second task was to introduce time into the city model. I started with the TRAFFIC_TIME class which counts the time. To use and test this class I developed the example application city_time. Furthermore, like the City3D application, my application is another example how to use the map widget. I added travelers to the Traffic library and then implemented their representation. This included modeling 3D objects which can be used in the object loader. The description of this section can be found in the third chapter 'Time and Travelers'.

The last chapter states some future work based on my project.

2. Map widget

**TRAFFIC 3D MAP WIDGET**

My first task was to extract the essentials of the implementation of the map in the TRAFFIC CITY 3D Project into a map widget. The widget can now be found in the tr_traffic.tr_visualization.tr_3d_visualization cluster. The TRAFFIC_3D_MAP_WIDGET class inherits from the EM_3D_COMPONENT class which enables it to be drawn in a EM_3D_COMPONENT_SCENE. This map widget class sets up all the parameters of the map such as the plane, the sunlight or the background. As a TRAFFIC_MAP can be loaded from an XML file this is done in this class. To represent buildings, lines, places or travelers this class initializes them with their representation class, respectively. A description of those classes follow in the next section.

As collision is globally in this map the collision polygons can be found here, but not all of them are implemented up to this point. If they are implemented they should be added here.

**REPRESENTATION CLASSES**

There are four classes that provide facilities to visualize elements of the map: TRAFFIC_LINE_REPRESENTATION for lines, TRAFFIC_PLACE_REPRESENTATION for places, TRAFFIC_BUILDING_REPRESENTATION for buildings, and TRAFFIC_TRAVELER_REPRESENTATION for visualizing travelers. Each representation class uses a factory to create its 3D objects and it implements the feature 'draw' which draws all the objects on the plane. Usually the objects are saved in an ARRAYED_LIST [EM_3D_OBJECT]. The representation classes have to give at least one decision procedure and an implementation of the 3D representation to their factories. This is done in the creation of the representation class just after the factories were created. To create an object in an add procedure we first have to call a decision procedure of the factory which then decides what function has to be called to get the 3D object. Moreover the collision polygons of the objects are created here, the map widget should get them from here.

**FACTORY CLASSES**

The factory classes are built as follows: there are gaugers and types. The creation functions are
saved as types and the decision procedures are saved as gaugers. The feature 'take_decision' gets the
decision out of the gaugers and saves the answer in the decision string. The feature 'create_object'
then calls the feature 'specify_object' which takes the decision and applies it to the types to get the
right function.

3. Time and Travelers

**Time**
The class TRAFFIC_TIME introduces time into the city model. It inherits from
EM_TIME_SINGLETON which is used to get delays. It provides two features to call procedures,
one of which is only used to call the 'take_tour' procedure after a certain interval. The other one
calls other procedures with other delays according to the time which is running. If someone wants
to call functions asynchronous he could use the inherited feature 'add_timed_callback'. Furthermore,
time is counted here and can be started, paused, resumed and reseted.

**Traveler**
The TRAFFIC_TRAVELER class represents travelers on the map. As a traveler is not a static
object, this class has a feature 'take_tour' which moves the traveler to the right coordinates if called
and gives the right angle to the traveler. The reason why this is done here is that this object carries
all the information about a specific passenger. Therefore also the type of a passenger is stored here,
which is of type TRAFFIC_TYPE. Only TRAFFIC_TYPE_WALKING and
TRAFFIC_TYPE_TRAM are used but others can be added easily.

**Traveler representation**
Like the other objects on the map also the travelers have their own representation container. The
feature 'draw' is provided here, too. But other than the other representations a traveler is added to a
map and the map has to be given as argument because there is no traveler in the xml file. A special
feature is the 'add_trams_per_line' which adds the given numbers to the line, or if a line doesn't
have enough stops the maximum of stops will be chosen. A TRAFFIC_TIME object is needed to let
the passengers walk as the procedure 'take_tour' from the TRAFFIC_TRAVELER object is inserted
to the reiterating procedures in TRAFFIC_TIME after a 3D object is created. 3D objects were
created by the following factory.

**Traveler factory**
In first attempts I implemented a factory the same way as the other ones described before. The
problem is that for such factories it is difficult to design good looking 3D objects. But the first
moving objects on the map were created out of this class but they looked only like coins on the
map. Because we wanted to have good looking objects I changed to a new factory.

**Object loader**
The object loader is now used as the factory for travelers. The decision is taken in the
TRAVELER_REPRESENTATION class. The main difference to the other factories is that this one
inherits from the EM_3D_OBJ_LOADER class and therefore loads *.obj files. Because the objects
first all took the same color as the last line drawn on the map, I had to change the color
representation such that the color is represented by a GL_VECTOR_3D [DOUBLE]. Files which
are read are closed in this class now but this will probably be fixed in the next version of the EM.
3D Objects

In order to produce *.obj files an open source editor was used. Because the parser for such files is not yet fully implemented in the EM library such an object is described only by vectors and 3 edge faces.

4. Future work

- The travelers do not have any collision boxes yet. They are needed as passengers should not walk through buildings or new traveler objects like cars should not collide.
- The travelers also don't know anything about the time. To implement time tables this is necessary.
- The map should be extended with depots where the trams can start.
- Streets are missing in the model up to now. When added a new traveler type which randomly travels on these streets should be added.
- There could be a possibility to travel through the city from a traveler's viewpoint.
The City_Time project implements a 3 dimensional visualization of a Traffic map file and adds time functionality.

This document is structured as follows:

- **Introduction**
  Welcome to City_Time!

- **Mouse controls**
  Shows the usage of City_Time application.
Welcome to *City_Time*!

The *City_Time* application displays traffic lines as colored polygons on a plane, adds passengers on this plane randomly, trams on their respective lines and provides mouse and keyboard support such that changing the viewpoint (zooming, different centres) is possible (easily). Moreover it runs time in the model.
Mouse controls

i) Mouse wheel: Controls the zooming factor, whereas the position stays invariant. There are two different zooming speeds depending on the distance to the plane.

ii) Mouse click: Is used to mark stations on the map. The origin can be marked with the left mouse button, the destination with the right one. If the user clicks on an empty spot, the marked stations are revoked.

iii) Mouse dragging:
   - Left mouse button: By dragging the mouse while pressing the left button one is able to translate the position of the map in the window, i.e. move the plane.
   - Right mouse button: By dragging the mouse while pressing the right button one is able to rotate the map around the the origin.

iv) Keyboard:
   - By pressing the up-/down- or left-/right- arrow keys the map will be rotated around the x- or y-axis respectively.
   - By pressing the enter key the translation will be reset to its initial value. However, zoom as well as rotation aren't affected.

Options
i) Start the time count. At the bottom of the toolbar you see the time going up. The trams (no changes possible) and the passengers start to move on the map. The simulated time is shown below and can vary from 5 to 60 minutes.

ii) Change the number of passengers to be displayed on the map: The number of passengers can be varied from 0 up to 1000. The passengers are randomly distributed.
The *City_Time* project implements a 3 dimensional visualization of a *Traffic* map file and adds time functionality.

This document is structured as follows:

- **Overview**
  - BON-Diagram to get an overview.
- **Widgets**
  - Description of the classes that are responsible for the user interface.
- **Time**
  - How does the time work?
- **Travelers**
  - How are the travelers created?
- **Notes**
  - Notes from the developer of *City_Time*. 

---

**Copyright © 2005, Michela Pedroni**

mailto:michela.pedroni@inf.ethz.ch

http://wiki.se.inf.ethz.ch/traffic/index.php/Main_Page

**Last Updated:** $Date: 2006/03/24 16:06:03 $
Bon diagram of all classes

INDEX >> CITYTIME DEVELOPER >> overview

Copyright © 2005, Michela Pedroni
mailto:michela.pedroni@inf.ethz.ch
http://wiki.se.inf.ethz.ch/traffic/index.php/Main_Page
Last Updated: $Date: 2006/03/12 10:17:05 $
City scene and 3D Map widget

CITY_3D_SCENE

Description

An instance of CITY_3D_SCENE gets created by the root class CITY_TIME_APPLICATION and called using the standard EM procedure. It uses the TRAFFIC_3D_MAP_WIDGET class to display the class and a toolkit from the EM library to create the toolbar.

Creation procedure

The creation procedure of the class creates all the needed widgets, such as slidebars or labels, and specifies their position, color and properties. If necessary, it subscribes a feature in the appropriate event handler. Moreover, it creates an instance of the CITY_3D_MAP class, which is responsible for all the things that have to be done in order to visualize the map.

Event handling

The event handling of this class is very important as it is the link between the user input and the features of the CITY_3D_MAP class that do the necessary operations such that the visual effects take place. It is important to notice that the instance of the CITY_3D_MAP class doesn't know its scene and therefore commands are only given from the scene to the map. If the map has to process information to the scene, such as the name of a marked point, the scene has to subscribe to the event handler of the map and query the desired information.

top

CITY_3D_MAP

Description

This is the main class of the project and has three main tasks. First it provides features for parsing the map XML file. Secondly, it is responsible for drawing the whole map with its lines, places, etc. This includes creating the necessary factories and drawing the map according to the enabled options. Last, it handles the user input. This means it has event handlers that change properties like zooming, translation or rotation (see User guide). Most of the implementation can be found in the class TRAFFIC_3D_MAP_WIDGET which is the parent of this class. Moreover this class handles the number of travelers on the map. It adjust the speed according to the simulated time.
TRAFFIC_TIME

Description

The class TRAFFIC_TIME is the representation of time in the model. The time supports starting time, pausing time, resuming time and resetting time. The feature time_count counts the time according to the simulated minutes. The functionality of the class EM_TIME_SINGLETON is extended by the feature add_callback_procedure which makes it possible not only to call functions every 'delay' time but also procedures. The feature add_callback_tour provides the possibility to handle the tours separately because it could be the case that one doesn't want to call the tours in the same frequency as the other procedures. Some generic extension could be thought of so that every procedure can have its one frequency.
TRAFFIC_TRAVELER

Description
The class TRAFFIC_TRAVELER provides all the information which the TRAVELER_REPRESENTATION needs to draw the traveler. There is the traffic_type which gives the type of the traveler, in this application only "tram" and "walking" type are used (descendants of the class TRAFFIC_TYPE). The feature take_tour is given to the TRAFFIC_TIME where it is used to get the current position of the traveler. There are three basic types of travelers: directed ones, where in the constructor an itinerary is given (ARRAYED_LIST [EM_VECTOR_2D]). Random ones which start at an origin (EM_VECTOR_2D) and fully random ones where origin, destination and speed are all given by random.

TRAFFIC_LINE_TRAVELER

Description
The class TRAFFIC_LINE_TRAVELER is a descendant from the class TRAFFIC_TRAVELER. As there is an obvious need for travelers which move on instances from class TRAFFIC_LINE, this class was created. It creates travelers of type "tram" if given a line. There is a comment in the code how to change this class such that the new objects have the same type as the line. Moreover in the class TRAFFIC_TRAVELER_REPRESENTATION the feature add_traveler has to be adapted. This should be a short task, but new *.obj files should be provided (there are only trams and passengers modeled up to now). If there is a need to set a line_traveler to a specific place there is the feature set_to_place which checks if the place is on the line and then places the traveler to that place. If one needs a place he can get it by calling the feature get_place which returns the place at 'number' in the feature last_place.

top

TRAFFIC_TRAVELER_REPRESENTATION

Description
The class TRAFFIC_TRAVELER_REPRESENTATION is responsible for distributing travelers of all kind on the plane. It stores all the travelers in a container and provides a feature to the map to draw all its objects.

Travelers distribution
There is always a traffic_traveler object associated to the 3D object which is drawn in this class, the position of each travelers is given by its corresponding traveler. Because of performance reasons the traveler stores its position according to the position on the map (the traveler uses the feature map_to_gl_coords by itself). As one should be able to change the map and its size this should be changed sometime.
Adding travelers

A 3D passenger object can be added to the map by calling the add_traveler feature with a corresponding TRAFFIC_TRAVELER object. As for the use of the traffic model the feature `add_trams_per_line` is provided which adds 'number' trams to the lines of 'a_map'. There can only be as many trams per line as there are stops.
External dependencies

This section lists some things that are kind of ugly in the sense that the application may not work perfectly if some external parameters are changed.

In **SHARED_CONSTANTS**:

- **Plane_size** is a constant that is chosen with the example maps in mind. This works for them but it might not work perfectly for other map files.
- **map_to_gl_coords** is also optimized for our example maps. It uses the range of the coordinates from the map xml file and centers the Zurich main station on the screen. This might not be intended for other examples.

These issues could be solved by either extending the map file or using a separate xml file.

Reutilisation and extensions

This section discusses some scenarios of how one could reuse this work.

1) Replacing the GUI:

As this is the intention of the class **TRAFFIC_3D_MAP_WIDGET** if someone wants to make a new application with a map then it should be inherited from the mentioned class which implements the basics. Therefore reusing is straightforward.

2) Extending displayed objects:

For all objects there are factories provided. If someone needs an object which differs from the default ones he could implement a function in the 
* _REPRESENTATION* classes and give it to the corresponding factory. Only the class **TRAFFIC_TRAVELER_REPRESENTATION**
doesn't support this way because its factory is different to the other ones. The corresponding factory **TRAFFIC_OBJECT_LOADER** inherits from the **EM_OBJ_LOADER** class which seems not fully developed. A new object has to be modeled in a * .obj file and simply loaded.

3) Extending functionality:

The object loader should get a new parser and some more functions to support different .obj files. Up to now only three edge faces are supported and no other features like surfaces which are possible in those files.

Ideas

- Promenade through the city: would make it possible to walk through the city with some traveler, which now exists, and to explore the city 'per pedes'.
- Schedules for the **TRAFFIC_LINE_TRAVELER** objects are desirable. Also delays of trams would give more realism to the simulation. Collision between two traveler objects can be detected and some action taken.
- In the xml there could be some hubs modelled from where all **line_travelers** start. There could be some automated start and creation of line traveler over the daytime.
The same is true for other travelers as for example rush hour could be modeled.

- Aside of tram and train lines there streets would make sense, and a new traveler type which travels along a random way, but always on the street.
This section consists of a description of the intention and use of the Traffic library with a description of each class and its simplified interface. The class overviews are sorted in alphabetic ascending order to be able to find a class description faster when knowing the name of it. For each class a short example of use will be added to show how it can be used. These examples can be found in the cluster test example.

This document describes how the Traffic library is structured and consists of the following parts:

**Map representation**

- **Overview**
  Gives an overview of the classes that are involved in the map representation part Traffic library

- **Classes**
  Describes each class of the map representation part Traffic library

**Visualization**

- **Overview**
  Gives an overview of the classes that are involved in the visualization part in the Traffic library

- **Classes**
  Describes each class of the visualization part Traffic library

**Input**

- **Overview**
  Gives an overview of the classes that are involved in the input part of Traffic library

- **Classes**
  Describes each class of the input part Traffic library
Let us start with a short overview of the Traffic library. The Traffic library was written to model a city and its public transportation system. Therefore you can imagine the library delivering you with all you need to build and work on a city map. This map can contain places with landmarks and public transportation lines. The library provides you with all you need and you even can get a tour through the city visiting all your places of interest. The overall model of the library is shown in the following figure.

As is visible from the picture, the most important class is the TRAFFIC_MAP class. It describes the map of a city and its public transportation system. Such a map consists of places and traffic transportation lines. Those lines in turn are made up of line sections. With this knowledge you can already build a map!
In the following sections the classes will be described. The layout for each class is: Requirements, Description, Class overview. In the Requirements the name of classes you need to know in order to understand the class described is given. Description is a short description of what the class does. It is more detailed than just the class description that you find in the class header. The class overview provides you with a class diagram.

- **TRAFFIC_BUILDING**
- **TRAFFIC_BUILDING_INFORMATION**
- **TRAFFIC_COLOR**
- **TRAFFIC_LINE**
- **TRAFFIC_LINE_TRAVELER**
- **TRAFFIC_LINE_SECTION**
- **TRAFFIC_LINE_SECTION_STATE**
- **TRAFFIC_LINE_SECTION_STATE_CONSTANTS**
- **TRAFFIC_MAP**
- **TRAFFIC_MAP_FACTORY**
- **TRAFFIC_MAP_LOADER**
- **TRAFFIC_PLACE**
- **TRAFFIC_PLACE_INFORMATION**
- **TRAFFIC_ROUTE**
- **TRAFFIC_SIMPLE_LINE**
- **TRAFFIC_TYPE**
- **TRAFFIC_TYPE_FACTORY**
- **TRAFFIC_TRAVELER**

**TRAFFIC_BUILDING**

**Requirements:**

**TRAFFIC_BUILDING_INFORMATION**

**Description**

The class **TRAFFIC_BUILDING** represents a building. A building is always a rectangle and is specified by its four corners (corner 1 is always the upper left, corner 2 the lower left, corner 3 the lower right and corner 4 the upper right one) and the three dimensions width, breadth and height. Furthermore, an angle (in range -70 to 70 degrees) can be specified by which the building is rotated.

A new building is created through the feature **make**, which needs the four corners of the building, its height and its name. The creation feature then calculates the width, the breadth and the center and sets the angle to zero.

In case the angle has to be changed after creation, this can be done through **set_angle**. If the building has to be drawn, then it needs also a unique identification number which can be set through **set_id**.

The four corners can be accessed through the four features: **corner1** (upper left corner), **corner2** (lower left corner), **corner3** (lower right corner) and **corner4** (upper right corner), whereas the three dimensions can be accessed through the tree features: **width**, **breadth** and **height**.
height. Finally, the angle can be accessed through `angle`.

There exists also the possibility to attach `TRAFFIC_BUILDING_INFORMATION` through `set_information`, which can afterwards be accessed through `information`.

A quite important feature is `contains_point`, which allows to check whether a point is inside a building. It is mainly used to check whether a building has been clicked.

**Class overview**

```
someDiagram
```

**TRAFFIC_BUILDING_INFORMATION**

**Requirements:**

None

**Description**

The class `TRAFFIC_BUILDING_REPRESENTATION` is a collection of additional information for a building. This can be a street, a house number and a textual description.

A street can be added by `set_street`, a house number by `set_house_number` and description by `set_description`.

**Class overview**

```
someMoreDiagrams
```

**TRAFFIC_COLOR**

**Requirements:**

None

**Description**

The class `TRAFFIC_COLOR` represents a RGB-Color. A RGB-Color is a color composed of the three additive components: red, green, blue. The three color parts can be accessed through the three features: `red`, `green`, `blue`. An important feature is `is_valid_color_part` which makes
sure only valid integer values for the rgb-parts are used. The object is created through the
**make** feature call which sets the color parts to the passed values. The individual color parts
can be changed through the **set_color** commands. Make sure the value you want to change
the color to is valid. For this the **is_valid_color_part** query can be used.

### Class overview

![Class overview diagram](image)

**TRAFFIC_LINE**

**Requirements:**

**TRAFFIC_COLOR, TRAFFIC_PLACE, TRAFFIC_LINE_SECTION, TRAFFIC_TYPE**

**Description**

The class **TRAFFIC_LINE** models a line of a public transportation system. Each line has a type,
e.g. **TRAFFIC_TYPE_BUS** for a bus line, **TRAFFIC_TYPE_RAIL** for a rail line and so on. This
type can be accessed through the feature **type**. In addition a line can have up to two
directions. A line which only has one direction is also valid. Each direction has a terminal
place.

A new line is created through the feature **make**. It takes as input the name of your new line
and a traffic type. The features **name, color, type, terminal_1, terminal_2** can directly be
called.

If you want to get the starting place of a direction of your line use the feature **start_to_terminal**. It takes as input either the place in **terminal_1** or **terminal_2**. If you use
just a place, make sure it is a terminal in a direction of the line. To make sure that a place is
a terminal the **is_terminal** query can be used. The features **one_direction_exists** and
**other_direction_exists** return true if one or other direction (depending on the feature you call)
exists. Only if a direction exists there exists a terminal and a starting place.

The query **is_valid_for_insertion** tells you if a line section, as it is, can be inserted into the
line. The query **is_valid_insertion** tells you if it is possible for a line section of the correct type
from a origin place to a destination place is a possible extension of the line in any direction.
So the argument in the second query (**is_valid_insertion**) is only the origin and destination of
a fictionary or real line section. **Extend** adds a line section to the line where it fits. So
directions can be extended at both their beginning and their end.

The color of a line can be changed or removed through the features **set_color** and
**remove_color**.

### Class overview
TRAFFIC_LINE_TRAVELER

Requirements:
None

Description
The class `TRAFFIC_LINE_TRAVELER` represents travelers which travel on a `TRAFFIC_LINE`. Now the default for the `traffic_type` is 'tram' but this should be changed in future implementations. There are two features added to this class: `get_place` and `set_to_place`. The first feature returns the place at the arguments position on the line, the result can be found at `last_place`. The second sets the traveler to the position of the given place, if the place is on the line.

Class overview

TRAFFIC_LINE_SECTION

Requirements:
`TRAFFIC_PLACE`, `TRAFFIC_LINE_SECTION`, `TRAFFIC_LINE_STATE`

Description
The class `TRAFFIC_LINE_SECTION` represents a connection of a line from one place to another. Those places are called origin and destination. A line section is of some traffic type, e.g. `TRAFFIC_BUS_TYPE`. A line section can be added to a line, forming the lines connection. It can belong to at most one line. Additionally a line section can have different states.

To create a new line section an origin place, a destination place and a traffic type have to be
defined. Such a line section will not belong to any line, its state will be set to some normal state and the polypoints are empty.

The feature length returns a calculated length of the line section. To calculate the length the polypoints are used, and if no polypoints exists, the position of the origin and destination place are used to calculate a length. This length becomes especially important when starting to calculate routes on lines and even a whole map of lines.

The changement of the attached line can only be carried out by a line. This is due to the fact, that when a line section has a line attached it should be in this line. Therefore the line is responsible to update line sections that are added to or removed from it.

Class overview

**TRAFFIC_LINE_SECTION_STATE**

Requirements:

**TRAFFIC_LINE_SECTION_STATE_CONSTANTS**

Description

The class TRAFFIC_LINE_SECTION_STATE provides the interface to define states and attach them to line sections. The available states are defined from the TRAFFIC_LINE_SECTION_STATE_CONSTANTS class.

A new state is simply created by calling its creation feature. The feature make always creates a normal state. To change it, call the feature set_state with one of the values defined in TRAFFIC_LINE_SECTION_STATE_CONSTANTS.

Class overview

**TRAFFIC_LINE_SECTION_STATE_CONSTANTS**

Requirements:

None
Description

The class `TRAFFIC_LINE_SECTION_STATE_CONSTANTS` defines all line section states and has the feature `is_valid_state_value` that tests any integer value to the states value and the feature `value_to_string` that returns a string representation of the state value.

Class overview

TRAFFIC_MAP

Requirements:

`TRAFFIC_LINE`, `TRAFFIC_PLACE`, `TRAFFIC_LINE_SECTION`

Description

The `TRAFFIC_MAP` is the collection of all lines, line sections, places and buildings. All elements are administrated by the map. Through the name of a place or a line you can retrieve it, you can search for line sections and even search shortest paths from one place to another. A map has a name and can additionally have a description that gives more information on the purpose of the map.

The name and the description can be accessed through the features `name` and `description`. Additionally a place of a given name can be accessed through the feature `place`. For all the elements of the map there exist features to find out whether there is such an element (`has_place`, `has_line` and `has_line_section`). To add new elements to the map use the features `add_place`, `add_line`, `add_line_section` or `add_building`. To remove all buildings from the map use `delete_buildings`.

Class overview
TRAFFIC_MAP_FACTORY

Requirements:

TRAFFIC_LINE, TRAFFIC_PLACE, TRAFFIC_LINE_SECTION, TRAFFIC_SIMPLE_LINE, TRAFFIC_MAP, TRAFFIC_TYPE, TRAFFIC_TYPE_FACTORY

Description

The TRAFFIC_MAP_FACTORY is used to create a map and its elements. The general principle is easy. Call the corresponding build feature. Test with the has-features if a valid object was created and access the last created object of a given element type with the corresponding query.

Create a new factory by calling it's make feature. This creates a new, blank factory. With the reset feature you can reset a factory to create a new map and it's elements from scratch.

At first you possibly want to build a map to be able to insert the map elements afterwards. The build feature that builds a named map is called build_map. The query map returns the last created map. With the call to the query has_map you make sure a valid map exists. After a successful call to this feature it is safe to call map. To build a place there are two features: to build a standard place whose position is at the origin (0,0) you call build_place; to build a place with another position call build_place_with_position. The features has_place and place have the same meaning as already mentioned generally. The building and accessing of a line section object works as with the other elements. The build features may look a little bit complicated with a long list of arguments, but the definition of a line section is quite large. A line section has an origin and destination place and a type. Additionally it can have polypoints defining its appearance and a line it belongs to. Therefore all these arguments have to be given to the build features of the line section.

Class overview

TRAFFIC_MAP_LOADER

Requirements:

TRAFFIC_MAP, TRAFFIC_SHARED_HANDLER

Description
The class `TRAFFIC_MAP_LOADER` is used to load maps, specified by its name.

A new instance of `TRAFFIC_MAP_LOADER` is created by calling `make`. It takes as input the name of the map to be loaded. The map is then loaded by `load_map`, which is either loaded from a dump file or, if this is not available or outdated, parsed from the specified XML file. If the map had to be parsed, a new dump is created, which can be used next time. Afterwards, the map can be accessed through `map`.

To detect changes in the XML file, a log file containing the XML file names and their timestamps is maintained.

**Class overview**

```
TRAFFIC_MAP_LOADER

map

TRAFFIC_MAP
```

**TRAFFIC_PLACE**

**Requirements:**

`TRAFFIC_PLACE_INFORMATION`

**Description**

The class `TRAFFIC_PLACE` represents a place in a city. It can have additional information like one or more pictures and a description attached to it. Additionally, it can have a position.

There exist two different ways to create a new place. The first way is to only define a name of the place, the position will be set to default (feature `make`). The second one is to also give its position to the creation feature (feature `make`).

The name, position, and information can be accessed through the according features. The information can be Void, whereas the name and position should never be void. The information and position can also be changed during the lifetime of a place object.

**Class overview**

```
TRAFFIC_PLACE

position

information

VECTOR_1D

TRAFFIC_PLACE_INFORMATION
```

**TRAFFIC_PLACE_INFORMATION**
Requirements:
LINKED_LIST

Description
The class TRAFFIC_PLACE_INFORMATION is a collection of additional information for a place. This can be one or more pictures (path to the pictures) and a textual description.

A picture is added with the feature extend_picture and a description is added with the command set_description. To delete a picture you have to know its path and use the command remove_picture. To remove a description just call remove_description.

Class overview

TRAFFIC_ROUTE
Requirements:
TRAFFIC_PLACE, TRAFFIC_LINE_SECTION

Description
The class TRAFFIC_ROUTE calculates the shortest path for a set of places you want to visit on a given map. You can change the route by adding or removing places you want to visit with the features extend and remove.

After calculating a route through the feature calculate_shortest_path the places and used line sections that are on the route can be accessed through places_on_route and line_sections.

Class overview

TRAFFIC_SIMPLE_LINE
Requirements:
TRAFFIC_LINE
Description

TRAFFIC_SIMPLE_LINE is a line that always has a line section in both directions. So if you add a line section from place A to place B the line section from place B to place A will be added as well. As a result you get a symmetric line.

The only feature that differs from the features of class TRAFFIC_LINE is the creation feature that has an additional argument: the traffic map that the simple line (and consequently its line sections) is in.

Class overview

TRAFFIC_TYPE

Requirements:
None

Description

The class TRAFFIC_TYPE is used to identify the type of a line or line_section. TRAFFIC_TYPE is the abstract class of all possible traffic types. Traffic types are mainly used to make sure that only line sections of a given type can be added to a line of some type. This is done to ensure that no bus drives on a railway and so on.

The traffic type classes have one interesting query: name returns a textual representation of the traffic type.

Class overview

TRAFFIC_TYPE_FACTORY

Requirements:
None
Description

The class `TRAFFIC_TYPE_FACTORY` creates singleton traffic types. This means you get the same type object every time you build a type. This makes it easier to compare two objects of a given traffic type: you can simply compare the references of the type. If they are identical the type of the two objects those type references belong to are the same.

The feature `valid_name` tests, if the name given as argument is a valid name of a traffic type. For valid names, traffic types can be built. The query `has_type` tells you whether the last call to build was successful.

The factory can be reset by the `reset` feature call. The most important feature of the factory is `build`. It generates a traffic type of the specified type. If you are not sure about the string, call `valid_name` to assure, that the type you want to build is a valid one. The last created type of the factory can be accessed with the call to `traffic_type`. The query `has_type` returns true if the type could be built.

Class overview

![Traffic Type Factory Diagram](image)

top

**TRAFFIC_TRAVELER**

Requirements:

None

Description

The class `TRAFFIC_TRAVELER` represents any type of traveler on the map. The actual type is then given through a `TRAFFIC_TYPE` object.

The feature `take_tour` lets the traveler change its position if called. There are two helper features: `set_coordinates` and `set_angle`, where the first one sets the new origin and destination, the second one set the angle_x which is used to let the traveler look into the right direction. If the traveler is random a new direction is given by the feature `give_random_direction` which sets the destination to a random position on the map.

There are several attributes like `traffic_type` which represents the type of the traveler, `traffic_info` could be used for information about the traveler, `speed` on the map which is changed according to `virtual_speed` and `time`, `is_reiterating` and `is_traveling_back` which is needed for the tour algorithm. There is the feature `index` which is a unique ID for each traveler.

Class overview
This is an overview over the 3D visualization widgets which are used to represent a map.

As everyone can see the main class is represented by the TRAFFIC_3D_MAP_WIDGET. There are some constants and functions which are used by all of the represented classes. These can all be found in the TRAFFIC_3D_CONSTANTS.
Traffic visualization classes

In the following sections the classes will be described. The layout for each class is: Requirements, Description, Class overview. In the Requirements section the name of classes you need to know in order to understand the class described is given. Description is a short description of what the class does. It is more detailed than just the class description that you find in the class header. The class overview provides you with a class diagram.

- **TRAFFIC_3D_CONSTANTS**
- **TRAFFIC_3D_MAP_WIDGET**
- **TRAFFIC_BUILDING_FACTORY**
- **TRAFFIC_BUILDING_REPRESENTATION**
- **TRAFFIC_LINE_FACTORY**
- **TRAFFIC_LINE_REPRESENTATION**
- **TRAFFIC_OBJECT_LOADER**
- **TRAFFIC_3D_PLACE_FACTORY**
- **TRAFFIC_PLACE_REPRESENTATION**
- **TRAFFIC_TRAVELER_REPRESENTATION**

### TRAFFIC_3D_CONSTANTS

**Requirements:**

None

**Description**

This class is shared by all other classes in this section. It has some default values like `window_width` or `place_height`. Another feature `map_to_gl_coords` can be found in this class. This feature translates given coordinates to coordinates according to the map.

**Class overview**

\[\text{ANY} \rightarrow \text{TRAFFIC_3D_CONSTANTS} \]

### TRAFFIC_3D_MAP_WIDGET

**Requirements:**

- **EM_3D_COMPONENT**
- **TRAFFIC_3D_CONSTANTS**

**Description**
The TRAFFIC_3D_MAP_WIDGET class is the main class in this cluster. As seen in the overview it contains all the representations of buildings, lines, places and travelers. It represents the whole map which can be plugged into an EM_3D_SCENE. It provides the feature draw that draws all the city elements onto the screen. The feature collision_polygons returns a list of polygons that allow to which is useful to search for a collision (e.g. used for placing buildings). The feature load_map is used to load from a file and create all the lines and places. There is a map dump which is loaded instead if there is such a map dump and this dump file is up to date.

There are three features to add buildings (add_building, add_buildings_randomly, add_buildings_along_lines) which either add one building, several randomly distributed buildings or several buildings along all traffic lines (expect railways). It is also possible to delete all the buildings with the feature delete_buildings. You can add a traveler through add_traveler. For all attributes there are corresponding setter methods.

It is also possible to get informed if a building is clicked. If a class needs this functionality, it has to subscribe to building_clicked_event.

**Class overview**

![](image)

**top**

TRAFFIC_BUILDING_FACTORY

**Requirements:**

None

**Description**

The class TRAFFIC_BUILDING_FACTORY is a factory for the 3D objects which represent the buildings on the map. The interface is for each of the factories the same: With the features add_gauger and remove_gauger one can handle procedures for decisions. These procedures can then be called through the feature take_decision which calls the right gauger and saves the decision in the string decision. For creating a representation of the built object one can add specific procedures through the features add_building_type and remove them by calling remove_building_type. The last two features are specify_object which takes the right building type according to the decision and create_object which is inherited from EM_3D_OBJECT_FACTORY and returns the created object.

**Class overview**
TRAFFIC_BUILDING_REPRESENTATION

Requirements:

TRAFFIC_BUILDING  TRAFFIC_3D_CONSTANTS

Description

The class TRAFFIC_BUILDING_REPRESENTATION is a container for all traffic building 3D objects. As expected there is a feature draw to draw all objects in the container. Buildings can be highlighted or un-highlighted by calling the features highlight_building or un_highlight_building. With the feature add_building a building can be added to the representation and delete_buildings deletes all the buildings from the representation.

Class overview

TRAFFIC_LINE_FACTORY

Requirements:

EM_3D_OBJECT_FACTORY

Description

The class TRAFFIC_LINE_FACTORY is a factory for the 3D objects which represent the lines on the map. The interface is for each of the factories the same: With the features add_gauger and remove_gauger one can handle procedures for decisions. These procedures can then be called through the feature take_decision which calls the right gauger and saves the decision in the string decision. For creating a representation of the built object one can add specific procedures through the features add_line_type and remove them by calling remove_line_type. The last two features are the feature specify_object which takes the right line type according to the decision and the feature create_object which is inherited from EM_3D_OBJECT_FACTORY and returns the created object.
**TRAFFIC_LINE_REPRESENTATION**

**Requirements:**
EM_3D_OBJECT TRAFFIC_3D_CONSTANTS

**Description**

The class **TRAFFIC_LINE_REPRESENTATION** is a container for all traffic line 3D objects. As expected there is a feature `draw` to draw all objects in the container. If a shortest line is added by `add_shortest_line` and not removed through `remove_shortest_line` the feature `draw_shortest_line` draws it. Lines can be highlighted or un-highlighted by calling the features `highlight_single_line`, `un_highlight_single_line` and similar ones. With the feature `add_lines` all lines of a map can be added, but single lines can be added too.

**TRAFFIC_OBJECT_LOADER**

**Requirements:**
EM_OBJ_LOADER

**Description**

In principle this class is a factory as well. It is used to create ".obj" files. The feature `set_em_color` sets the color of a created object. The feature `load_file` is used to load a new file, other than the original feature this feature closes it opened files after use. The feature `specify_object` specifies the object according to the loaded file and the given color.
TRAFFIC_PLACE_FACTORY

Requirements:  
EM_3D_OBJECT_FACTORY

Description  
The class TRAFFIC_PLACE_FACTORY is a factory for the 3D objects which represent the places on the map. The interface is for each of the factories the same: With the features add_gauger and remove_gauger one can handle procedures for decisions. These procedures can then be called through the feature take_decision which calls the right gauger and saves the decision in the string decision. For creating a representation of the built object one can add specific procedures through the features add_place_type and remove them by calling remove_place_type. The last two features are the feature specify_object which takes the right place type according to the decision and the feature create_object which is inherited from EM_3D_OBJECT_FACTORY and returns the created object.

TRAFFIC_PLACE_REPRESENTATION

Requirements:  
TRAFFIC_3D_CONSTANTS

Description  
The class supports the feature draw to draw all places. The feature highlight_place highlights a place. Through the feature add_places all places of a map are added to the container.
Requirements:

TRAFFIC_3D_CONSTANTS

Description

Like all other containers the feature draw is also provided to draw all travelers. To add a traveler call add_traveler. As there are no travelers provided in the map you have to provide the TRAFFIC_MAP to add this traveler. The feature remove_traveler removes a walking traveler from the map, the feature remove_specific_traveler does the same by comparing the given traveler with all others in the container. If found it is removed. There is a feature add_trams_per_line which adds the given number or maximal trams to each line.

Class overview

Copyright © 2005, Michela Pedroni
mailto:michela.pedroni@inf.ethz.ch
http://wiki.se.inf.ethz.ch/traffic/index.php/Main_Page
Last Updated: $Date: 2006/03/24 13:36:41 $
This is an overview over the input part, which is used to load from XML files.
Traffic input classes

In the following sections the classes for the map loading will be described. The layout for each class is: Requirements, Description, Class overview. In the Requirements section the name of classes you need to know in order to understand the class described is given. Description is a short description of what the class does. It is more detailed than just the class description that you find in the class header. The class overview provides you with a class diagram.

- **TRAFFIC_BUILDING_PARSER**
- **TRAFFIC_BUILDING_NODE_PROCESSOR**
- **TRAFFIC_BUILDINGS_NODE_PROCESSOR**

**TRAFFIC_BUILDING_PARSER**

Requirements:

TRAFFIC_XML_INPUT_FILE_PARSER, TRAFFIC_NODE_PROCESSOR_REGISTRY, TRAFFIC_BUILDING_NODE_PROCESSOR, TRAFFIC_BUILDINGS_NODE_PROCESSOR

Description

This class is used for parsing an XML file that contains building information and for processing the received data so the buildings specified appear on the map. The XML file has to follow the following DTD structure:

```xml
<!ELEMENT buildings (building*)>
<!ELEMENT building EMPTY>
<!ATTLIST building
  name CDATA #REQUIRED
  x1 CDATA #REQUIRED
  y1 CDATA #REQUIRED
  x2 CDATA #REQUIRED
  y2 CDATA #REQUIRED
  height CDATA #REQUIRED
  angle CDATA #REQUIRED
>
```

The building position is specified by the upper left corner (x1, y1) and the bottom right corner (x2, y2) of the building. One has to be aware that the x-axis increases in the left direction and the y-axis increases in the up direction.

A new building parser is created through the feature `make_with_map` which needs a TRAFFIC_3D_MAP_WIDGET as input.

The file can then be added through the inherited feature `set_file_name` and parsed through (the also inherited) feature `parse`. If the parsing is completed successfully (checked `has_error`) processing can be started with `process`. The processing is based on the rules specified in TRAFFIC_BUILDING_NODE_PROCESSOR and TRAFFIC_BUILDINGS_NODE_PROCESSOR.
Class overview

TRAFFIC_BUILDING_PROCESSOR

Requirements:

TRAFFIC_NODE_PROCESSOR

Description

In this class is specified how TRAFFIC_BUILDING_PARSER has to process a building element. It has specified the element name (name) and the mandatory attributes (mandatory_Attributes).

In the feature process is specified, what has to be done during processing. In the case of a building element, we only have to check whether the required attributes are supported and have proper type and afterwards create a new building.

Class overview

TRAFFIC_BUILDINGS_PROCESSOR

Requirements:

TRAFFIC_NODE_PROCESSOR

Description

In this class is specified how TRAFFIC_BUILDING_PARSER has to process a buildings element. It has specified the element name (name) and the mandatory attributes (mandatory_Attributes).

In the feature process is specified, what has to be done during processing. In the case of a buildings element, we to process all subnodes.