Traffic 3.0
Realistic Buildings And Performance

Semester Thesis

By: Fabian Wüst
Supervised by: Michela Pedroni
Prof. Bertrand Meyer

Student Number: 01-907-575
# Table of contents

1. Introduction
2. Implementation of buildings
   2.1 Modeling buildings
   2.2 Visualizing buildings
   2.3 Parsing input files
   2.4 Example application City3D
3. Improvement of map loading
   3.1 Error handling
   3.2 Map parsing
4. Future tasks
5. Online documentation
   5.1 City3D
   5.2 Traffic library
   5.3 Traffic visualization library
   5.4 Traffic input library
1. Introduction

The goal of this project was to introduce realistic buildings to the traffic library and show how they can be used by extending the existing City3D application. Therefore, a reimplementation of the building part of the City3D application had to be done. Since the buildings in the existing City3D application were not part of the library and were merely drawn on top of the map rather than included in the map into the Traffic library, the building part needed to be redesigned and adapted for inclusion into the Traffic library.

2. Implementation of buildings

2.1 Modelling buildings

One of the goals of my project was to make buildings clickable. For this purpose a building representation and a data structure that allows checking, whether a mouse click occurred over a building was needed.

To improve the performance, buildings are represented through their rectangular base (and therefore their four corners P1, P2, P3, P4), their height and angle (in rage -70 to 70 degrees) which allows a building to be rotated around its center.

Be aware that P1 always needs to be the upper most corner and that the x-axis increases to the left hand side (which is due to the fact, that Open GL uses this coordinate system)
The requirements on the data structure to be used for storage of the buildings were quite demanding.
On one hand, we wanted fast insertion and iteration to allow insertion of large numbers of buildings. On the other hand, search performance to detect clicks on a specific building needed to be satisfied as well.

To meet both requirements, the city plane is split into four quadrants and a linked list is maintained for each of them. A building is contained in a list, if it has at least one corner inside the corresponding quadrant. Therefore, if a building is spread over multiple quadrants, it is contained in each of the corresponding lists.

This allows searching approximately only a fourth of the buildings (under the assumption that the buildings are uniformly distributed over the whole plane).

Since a building is represented through a rectangle (and therefore its four corners) and an angle, it is quite easy to check, whether a given point lies inside it.

A point \( P(x,y) \) is only inside the rectangle if
- \( P3.y < y < P1.y \) and
- \( P4.x + dx2 < x < P2.x - dx1 \)

with
- \( dx1 = \frac{abs(y - P2.y)}{tan(a)} \)
- \( dx2 = \frac{abs(P4.y - y)}{tan(a)} \)
2.2 Visualizing buildings

To be able to draw buildings fast, I decided to use a common simple representation (picture on the right) which is stored in a displaylist and can be scaled, translated or rotated if needed.

2.3 Parsing input files

Another goal was to load buildings from a file. The chosen file format is XML and files must conform to the following DTD:

```
<?xml version='1.0'?>
<!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
>
```

A building is specified through the upper left and lower right corner, its height and an angle.

The xml parser for the buildings reuses existing functionality of the map loading mechanism of Traffic. The classes TRAFFIC_BUILDING_PARSER, TRAFFIC_BUILDING_PROCESSOR and TRAFFIC_BUILDINGS_PROCESSOR are newly implemented classes. For more information see section 5.4.
2.4 Example application City3D

The City3D application serves as both, a test bed to check the newly written classes and features, and as an example for their usage. To show the possibilities of the new buildings, the existing application got extended with functionalities to add buildings to the city. They can be added randomly, loaded from a XML file or distributed along tram lines and streets. The last option supports the most realistic view since in real cities it is quite common to build houses along streets. Moreover, it is now possible to mark buildings by clicking on them and to show their names.

3. Improvement of map loading

3.1 Error handling

A subtask of my work was to improve the error handling during map loading, since the library did actually recognize errors during parsing and throw an exception, but did not give any graphical output nor catch the exceptions. The new implementation catches the exceptions and then shows the error message that is supported by the XML parser.

3.2 Map loading

One of the weakest points of the original Traffic library was the slow map loading mechanism. In order to improve it, the object structure of the Traffic map model is now dumped to a file, so that it can be read from there the next time it is used. This of course can only be done, if the original XML file was not changed in the meantime.
To detect changes in the XML file, a log file containing the XML file names and their timestamps is maintained.

4. Future tasks

An interesting task would be to import the real buildings from Zurich to the map, which would make City3D even more realistic. This could be done quite easily, since the possibility to include buildings does already exist. Furthermore, the way the buildings are internally stored could be improved in order to speed up building creation and clicking detection.
Welcome to City3D!

The City3D application displays traffic lines as colored polygons on a plane, adds buildings and provides mouse and keyboard support such that changing the viewpoint (zooming, rotation, different centers) and interaction with the buildings is possible. Moreover it provides different light settings and visualizes shortest paths in the city.
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 are not affected.

Options

Choose a map:

map/paris.xml

Load map

Zoom in  Zoom out

Number of buildings:

0

- Transparent buildings
- Show sun
- Show coordinates
- Highlight lines
- Show buildings
- Shortest path

Marked station:
i) Choose the map to be loaded and load it.

ii) Zoom in and out.

iii) Change the number of buildings to be randomly displayed on the map: The number of buildings can be varied from 0 up to 4900.

iv) Make the buildings transparent in order to achieve a better overview.

v) Change lighting of the map: Initially, the scene is lit by a constant light coming down the y axis. However, this can be changed to a variable light that tries to imitate the effect of the sun.

vi) Show a coordinate system: This shows a coordinate system at the origin for better orientation.

vii) Highlight the traffic lines by drawing them with different height on the map, in order to achieve a better overview.

viii) The buildings can be hidden or shown by enabling/disabling the "show buildings" option.

ix) Show the shortest path between two marked stations and display it as a white line. The origin for the route is marked by clicking on the place with the left mouse button and the destination with the right mouse button.

x) When two stations are selected, a new button appears. If you press it, you can take a traffic line ride in a first-person perspective between the two marked stations.

Name of marked building:
Hauptbahnhof

Choose building file:
buildings/HR.xml

Load buildings from file

Load buildings along lines

Delete buildings

i) Shows the name of the most recently marked building.

ii) Choose the XML file for building loading and load them.

iii) Load buildings along all traffic lines (expect railway).

iv) Delete all buildings.

top
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.

- **TRAFFICBUILDING**
- **TRAFFICBUILDING_INFORMATION**
- **TRAFFICCOLOR**
- **TRAFFICLINE**
- **TRAFFICLEINETRAVELER**
- **TRAFFICLEINSECTION**
- **TRAFFICLEINSECTIONSTATE**
- **TRAFFICLEINSECTIONSTATE_CONSTANTS**
- **TRAFFICMAP**
- **TRAFFICMAP_FACTORY**
- **TRAFFICMAP_LOADER**
- **TRAFFICPLACE**
- **TRAFFICPLACE_INFORMATION**
- **TRAFFICROUTE**
- **TRAFFICSIMPLELINE**
- **TRAFFICTYPE**
- **TRAFFICTYPE_FACTORY**
- **TRAFFICTRAVELER**

**TRAFFICBUILDING**

**Requirements:**

**TRAFFICBUILDING_INFORMATION**

**Description**

The class **TRAFFICBUILDING** 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. 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

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

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

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

![Diagram of TRAFFIC_LINE, TRAFFIC_LINE_SECTION, TRAFFIC_TYPE, TRAFFIC_LINE_SECTION_STATE, TRAFFIC_PLACE, VECTOR_2D connections]

**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**.

---

**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

![Diagram of `TRAFFIC_LINE_SECTION_STATE_CONSTANTS` class]

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

![Diagram of traffic map class relationships]
**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 its **make** feature. This creates a new, blank factory. With the **reset** feature you can reset a factory to create a new map and its 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 Class Diagram]

**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**

![Diagram](image1)

**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**

![Diagram](image2)

**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 way 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
   + Bus_type
   + Rail_type
   + Street_type
   + Taxi_type
   + Walking_type
```

**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 it's 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

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**

```
+ TRAFFIC_3D_MAP_WIDGET
  * TRAFFIC_3D_CONSTANTS
  * EM_3D_COMPONENT
  * DOUBLE_MATH
```

**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

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

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.

Class overview

TRAFFIC_OBJECT_LOADER

Requirements:
EM_OBJ_LOADER

Description
In principle this class is a factory as well. It is used to create ".obj" files. As for now it is not possible to make one factory and make several loads. Therefore, everytime a 3D object should be created a new factory has to be created too. 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 is opened files after use. The feature specify_object specifies the object according to the loaded file and the given color.

**Class overview**

**TRAFFIC_PLACE_FACTORY**

**Requirements:**

None

**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.

**Class overview**

**TRAFFIC_PLACE_REPRESENTATION**

**Requirements:**

None

**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.
TRAFFIC_TRAVELER_REPRESENTATION

Requirements:
None

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.
This is an overview over the input part, which is used to load from XML files.
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.