Traffic v3.2
Improving Random Building Placement

Semester Project

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

Student Number: 02-806-677
Table of Contents

1. Introduction

2. Background
   2.1. Random building placement so far

3. Improving the Algorithm
   3.1. Concepts
      3.1.1. Speed and Correctness
      3.1.2. Realisticity
   3.2. Implementation
      3.2.1. Speed and Correctness
         3.2.1.1 Discretization
         3.2.1.2 Marking Cells
      3.2.2. Realisticity
      3.2.3. Details and Remarks

4. Possible future Work

5. Conclusions

6. Appendix
   6.1. Modified Documentation
Introduction

The goal of this project was to provide the traffic library with a reliable and correct building placement feature for three dimensional maps. Enough buildings should be placed for the city to have a realistic look. Buildings should be placed along roads and traffic lines. The correctness should be ensured. No buildings should overlap with roads, lines or other buildings.

Background

Random building placement so far
In the original version of traffic, the class TRAFFIC_3D_MAP_WIDGET provided the features: add_buildings_randomly and add_buildings_along_lines. They worked as follows:

```plaintext
add_buildings_randomly (n)
   until n buildings placed loop
      generate a building at a random location
      if not has_collision(building) then
         place the building
      end
   end

add_buildings_along_lines
   for each line or road
      for each possible position on left side of the line or road
         generate building
         if not has_collision(building) then
            place the building
         end
      end
   end
```

'has_collision' was evaluated by testing the collision of the generated building with all lines, roads or other buildings on the map. This was done using bounding surfaces called collision polygons. Every line, road, or building had a corresponding collision polygon. Although these bounding surfaces were rectangles, they were tested as if they were circles. I suppose this was done for performance reasons.
has_collision(building)
   for each line or road collision polygon
      check for collision
   end
   for each other building collision polygon
      check for collision
   end

The following snapshot visualizes the collision polygons of the roads and lines.

I found the above algorithms to be quite clever, but they had massive performance drawbacks. As I discovered later during my project, add_buildings_randomly wasn't correct. When a large number buildings was placed, buildings started overlapping. Additionally, the termination of add_buildings_randomly wasn't ensured.
Improving the Algorithm

I wanted to write an algorithm which was faster than the above two and that would generate a more realistic, correct city.

Concepts

Speed and Correctness
The main point of improvement was the has_collision function. Every line, road, or other building was tested against the building to be placed, even if it was at a completely different place on the map. To check if a building can be placed, only the area directly under the building should be tested.

Realisticity
I thought about how a human being would draw a city if he were given pencil and paper. The roads would already be drawn on the paper. I came to the conclusion, that he’d first draw buildings along the roads. Then, he’d fill up the remaining space with buildings. These buildings would have similar directions to the buildings along the lines. I decided to try and imitate this method.

Implementation

Speed and Correctness
Discretization
In search of the solution, I was inspired by the way the developers of SimCity or Civilization had mastered a similar problem. The idea was to discretize the map into small squares, or cells. Any object on the map would occupy a certain number of cells, depending on its size.
I decided to use a simple two dimensional, boolean array to represent the cells:

boolean_grid: ARRAY2[BOOLEAN]

This construct is completely independent from the rest of the map. The only connections between the map and boolean_grid are the features: grid_coordinate and gl_coordinate. grid_coordinate returns the cell number of the boolean grid for a point on the map. gl_coordinate returns the map coordinate for a certain grid cell number. The following snapshot shows a sample grid with a size of about 200.
Marking Cells

I needed a feature to mark all cells which are occupied by roads and lines. This is handled by the feature mark_occupied. This feature traverses all roads and lines, and calls a standard line drawing algorithm for each road- or line section. This line drawing algorithm marks the cells to occupied along a road- or line section, and also takes into account the width of the line.

mark_occupied needs to be called only once, before one can start placing buildings. This results in a major performance gain. Before, an algorithm of similar complexity had to be called for every randomly generated building, independently if it could be placed or not. The following snapshot shows the cells which were marked occupied in light green.
add_buildings_along_lines and add_buildings_randomly now simply check, if the cells of the rectangular area beneath a building are all free. If yes, the building is placed, and the cells are marked occupied. For rotated buildings, a quadratic cell cluster with the length of the building diagonal is checked and marked. This is done for simplicity and performance reasons.

**Realisticity**
To achieve the realisticity in the sense of a city drawn by a human being, I decided to simply combine the existing algorithms: add_buildings_randomly and add_buildings_along_lines

So my place_buildings feature would look like this:

```java
place_buildings
    add_buildings_along_lines
    add_buildings_randomly
```

However, I decided to modify the add_buildings_randomly function. I added a max_iterations integer to the map to ensure its termination. I also added 3 parameters: size_of_plane, a_template and a_randomizer. size_of_plane defines the size of the area in which the buildings are placed. The center of the area is the center of the map. a_template defines which type of building is placed. a_randomizer is an instance of type RANDOM. Before my modification this was a local variable of the feature. It was used to generate the random building positions. However, the generated positions were always the same, because always the same seed was set.

These modifications give the caller of add_buildings_randomly more freedom which buildings he wants to place, an where he wants to place them. This feature is currently used to place a few skyscrapers in the middle quarter of the map:

```java
place_buildings
    add_buildings_along_lines
    add_buildings_randomly(n, planesize/2, skyskraper, randomizer )
    for all existing templates
        add_buildings_randomly(n, planesize, template, randomizer )
    end
```
Details and Remarks

Almost all added features are located in the class `TRAFFIC_3D_MAP_WIDGET` of the traffic library. Only a few modifications and additions needed to be made to other classes. The only publicly accessible feature is `place_buildings_randomly`. All other features are concealed within `TRAFFIC_3D_MAP_WIDGET`. I chose to do this for consistency reasons. A traffic developer may want to experiment with the size of the grid: `grid_size`. I for myself have come to the conclusion that sizes between 300 and 800 are most preferable. Large grid sizes negatively affect performance, but buildings are placed closer to each other. A traffic developer may also want to experiment with the `place_buildings` function. With successive calls of `add_buildings_randomly`, many layers of buildings may be placed, distributed over the area length specified.

The collision polygons for each placed building are still generated, and may be used normally. However, the random building placement algorithm has no need for them. I added a building density parameter to `place_buildings_randomly`. One can choose between three degrees of building density. The following snapshot has a density of two.

Possible future Work

One may want to make the access to the grid more publicly. For example, a future student could add public grid manipulation features, or export existing ones from `TRAFFIC_3D_MAP_WIDGET`. It is a much more efficient concept than the collision
polygons. So it would be useful for performance aware applications. The grid was designed to be used only for random building placement. For now, it is only used by the features: place_buildings_randomly and its feature calls. It is important to ensure that the consistency is maintained when exporting features of the grid from TRAFFIC_3D_MAP_WIDGET.

The feature add_buildings_randomly could be extended with an additional parameter to set the placement angle of the corresponding layer of buildings. That way one could achieve a larger diversity of building orientations.

One may also want to investigate the reason, why the old version of add_buildings_randomly was incorrect for many buildings. There might be a connection to the generation or collision detection of the collision polygons.

Conclusions

I compared the efficiency of the features which existed before with the new feature for the map 'Paris'. The following are execution times of the new feature on my laptop:

Grid Size 300
- load map and building templates 16.3 seconds
- mark occupied positions 5.0 seconds
- add buildings along lines 20.7 seconds
- add 630 buildings randomly 29.5 seconds
- Total 1 minute 11.5 seconds

Grid Size 800
- load map and building templates 16.3 seconds
- mark occupied positions 11.4 seconds
- add buildings along lines 22.4 seconds
- add 630 buildings randomly 30.4 seconds
- Total 1 minute 22.5 seconds

The total execution time for a similar setup using the old features was 2 minutes 1.9 seconds. This time can be divided as follows:

- load map and building templates 16.3 seconds
- add buildings along lines 47.8 seconds
- add 630 buildings randomly 57.8 seconds

However, the comparability of the new feature and the old ones is limited. The new feature is correct, the old ones weren't.

I was very happy with the results. The access to the boolean grid turned out to be very efficient. A large performance gain could be achieved. The number of buildings which can be placed has drastically increased. The buildings can be placed closer than before. I
believe that the increased number of buildings on the map, and their close positioning are the main gains to realism. The orientation of the buildings along the roads and lines plays a minor part. The building placement is correct, no buildings overlap with roads, lines, or other objects.
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_BUILDING_FACTORY, TRAFFIC_3D_BUILDING_SIMPLE_FACTORY, TRAFFIC_3D_BUILDING_FANCY_FACTORY
- TRAFFIC_3D_BUILDING_REPRESENTATION
- TRAFFIC_3D_CONNECTION_FACTORY, TRAFFIC_3D_CONNECTION_HIGHLIGHTED_REP_FACTORY, TRAFFIC_3D_CONNECTION_LINE_REP_FACTORY
- TRAFFIC_3D_CONSTANTS
- TRAFFIC_3D_DRAWING_PRIMITIVES
- TRAFFIC_3D_LINE_REPRESENTATION
- TRAFFIC_3D_MAP_WIDGET
- TRAFFIC_3D_OBJ_LOADER
- TRAFFIC_3D_PATH_REPRESENTATION
- TRAFFIC_3D_PLACE_FACTORY, TRAFFIC_3D_PLACE_CIRCLE_REP_FACTORY, TRAFFIC_3D_PLACE_SQUARE_REP_FACTORY
- TRAFFIC_3D_PLACE_REPRESENTATION
- TRAFFIC_3D_ROAD_REPRESENTATION
- TRAFFIC_3D_SUN_REPRESENTATION
- TRAFFIC_3D_TEXTURE_OBJECT
- TRAFFIC_3D_TEXT_SCANNER
- TRAFFIC_3D_TRAVELER_REPRESENTATION

TRAFFIC_3D_BUILDING_FACTORY, TRAFFIC_3D_BUILDING_SIMPLE_FACTORY, TRAFFIC_3D_BUILDING_FANCY_FACTORY

Requirements:

EM_3D_OBJECT_FACTORY, TRAFFIC_3D_OBJ_LOADER

Description

The class TRAFFIC_3D_BUILDING_FACTORY is a deferred factory for the 3D objects which represent the buildings on the map. At the moment two factories are effective descendants of this class: TRAFFIC_3D_BUILDING_SIMPLE_FACTORY and TRAFFIC_3D_BUILDING_FANCY_FACTORY. The class TRAFFIC_3D_BUILDING_REPRESENTATION uses one of these factories to generate new building views. Through a call to create_object which is inherited from EM_3D_OBJECT_FACTORY a new 3d object is created and returned.

Class overview
TRAFFIC_3D_BUILDING_REPRESENTATION

Requirements:
TRAFFIC_BUILDING, TRAFFIC_3D_CONSTANTS, TRAFFIC_BUILDING_FACTORY

Description

The class TRAFFIC_3D_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 unhighlight_building. With the feature add_building a building can be added to the representation and delete_one_building respectively delete_buildings deletes one specific respectively all the buildings from the representation.

Class overview

TRAFFIC_3D_CONNECTION_FACTORY,
TRAFFIC_3D_CONNECTION_LINE_REP_FACTORY,
TRAFFIC_3D_CONNECTION_HIGHLIGHTED_REP_FACTORY

Requirements:

EM_3D_OBJECT_FACTORY

Description

The class TRAFFIC_3D_CONNECTION_FACTORY is a factory for the 3D objects which represent the connections (be it roads, line sections or paths) on the map. Additionally, to the features inherited from EM_3D_OBJECT_FACTORY (such as create_object which creates a new 3d object) the class TRAFFIC_3D_CONNECTION_FACTORY always also creates a collision polygon for a new 3d object. To specify for which connection a new 3d object needs to be created use set_connection.

Class overview

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_DRAWING_PRIMITIVES**

**Requirements:**

GL_FUNCTIONS, GLU_FUNCTIONS, EM_CONSTANTS

**Description**

The class TRAFFIC_3D_DRAWING_PRIMITIVES provides basic features for drawing 3d objects such as create_circle, create_cube, and create_cylinder.

**Class overview**

---

**TRAFFIC_3D_LINE_REPRESENTATION**

**Requirements:**

EM_3D_OBJECT, TRAFFIC_3D_CONSTANTS

**Description**

The class TRAFFIC_3D_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. Lines can be highlighted or un-highlighted by calling the features highlight_single_line and un_highlight_single_line. With the feature add_lines all lines of a map can be added, but single lines can be added too. TRAFFIC_3D_LINE_REPRESENTATION uses descendants of TRAFFIC_3D_CONNECTION_FACTORY to create the 3d objects.

**Class overview**
**TRAFFIC_3D_MAP_WIDGET**

**Requirements:**

EM_3D_COMPONENT, TRAFFIC_3D_CONSTANTS, TRAFFIC_BUILDING_REPRESENTATION, TRAFFIC_PLACE_REPRESENTATION, TRAFFIC_PATH_REPRESENTATION, TRAFFIC_LINE_REPRESENTATION, TRAFFIC_TRAVELER_REPRESENTATION

**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 which is useful for searching for a collision (e.g. used for placing buildings). The feature set_map is used to set the map to be displayed and creates all the visual representation needed for display.

There is a feature to place buildings randomly (place_buildings_randomly) which places several randomly distributed buildings on the map. The buildings are first placed along the line sections, then on the rest of the map. A two dimensional boolean grid is used to check for collisions. It is possible to delete all the buildings with the feature delete_buildings.

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_OBJ_LOADER

Requirements:

EM_3D_OBJECT_FACTORY

Description

This class is provides facilities to load OpenGl 3d objects from *.obj files (Wavefront object files). It uses caching to optimize the recurring loading of such files.

Class overview

TRAFFIC_3D_PATH_REPRESENTATION

Requirements:

EM_3D_OBJECT, TRAFFIC_3D_CONSTANTS, TRAFFIC_3D_CONNECTION_FACTORY

Description

The class TRAFFIC_3D_PATH_REPRESENTATION is a container for all traffic path 3D objects. As expected there is a feature draw to draw all objects in the container. TRAFFIC_3D_PATH_REPRESENTATION uses descendants of TRAFFIC_3D_CONNECTION_FACTORY to create the 3d objects.

Class overview
TRAFFIC_3D_PLACE_FACTORY, TRAFFIC_3D_PLACE_CIRCLE_REP_FACTORY, TRAFFIC_3D_PLACE_SQUARE_REP_FACTORY

Requirements:
EM_3D_OBJECT_FACTORY

Description
The class TRAFFIC_3D_PLACE_FACTORY is a deferred factory for the 3D objects which represent the places on the map. Additionally, to the features inherited from EM_3D_OBJECT_FACTORY (such as create_object which creates a new 3d object) the class TRAFFIC_3D_PLACE_FACTORY always also creates a collision polygon for a new 3d object. To specify for which connection a new 3d object needs to be created use set_connection.

TRAFFIC_3D_PLACE_CIRCLE_REP_FACTORY and TRAFFIC_3D_PLACE_SQUARE_REP_FACTORY

Class overview
TRAFFIC_3D_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.

Class overview

TRAFFIC_3D_ROAD_REPRESENTATION
**Requirements:**

**EM_3D_OBJECT TRAFFIC_3D_CONSTANTS**

**Description**

The class TRAFFIC_3D_ROAD_REPRESENTATION is a container for all traffic road 3D objects. As expected there is a feature draw to draw all objects in the container. In the feature draw each type of road is associated to a particular representation on the map. TRAFFIC_3D_ROAD_REPRESENTATION uses descendants of TRAFFIC_3D_CONNECTION_FACTORY to create the 3D objects.

**Class overview**

![Diagram](image)

**TRAFFIC_3D_SUN_REPRESENTATION**

**Requirements:**

**TRAFFIC_SHARED_TIME**

**Description**

The class TRAFFIC_3D_SUN_REPRESENTATION provides the facilities to display a sun that is moving according to the current simulated time in the system. Therefore, it uses TRAFFIC_SHARED_TIME.

**Class overview**
**TRAFFIC_3D_TEXTURE_OBJECT**

**Requirements:**

**EM_3D_OBJECT**

**Description**

The class TRAFFIC_3D_TEXTURE_OBJECT is an extended EM_3D_OBJECT, i.e. an OpenGL displaylist that can have textures (bitmaps that are wrapped around the skeleton object).

**Class overview**

**TRAFFIC_3D_TEXT_SCANNER**

**Requirements:**

None

**Description**

The class TRAFFIC_3D_TEXT_SCANNER is a string tokenizer used for reading *.obj files (see class TRAFFIC_3D_OBJ_LOADER).

**Class overview**
TRAFFIC_3D_TRAVELER_REPRESENTATION

Requirements:

TRAFFIC_3D_CONSTANTS, TRAFFIC_3D_OBJ_LOADER

Description

Like all other containers the feature draw is also provided to draw all travelers. To add a traveler call add_traveler. As factory for 3d objects it uses TRAFFIC_3D_OBJ_LOADER.

Class overview