Sim-city-1, ELEC-A7150
C++ Programming

15.12.2017

Aalto University, School of Electrical Engineering

Documentation

Harti Hänninen

Risto Koverola

Ilari Rauhala

Tianzhong Pan

Palautettu: 15.12.2017

Table of Contents

Table of Contents 2
1. Overview 4
2. Software structure 5
2.1. Class structure 5
2.1.1. main 6
2.1.2. graph 6
2.1.3. vertex 6
2.1.4. edge 6
2.1.5. vehicle 6
2.1.6. texturemanager 6
2.1.7. #### tools 6
3. Instructions for building and using the software 7
3.1. How to use the software 7
3.2. Use of C++ features 8
3.2.1. #### Containers 8
3.2.2. Smart Pointers 8
3.2.3. Exception handling 8
3.2.4. Rule of three 8
3.2.5. Dynamic binding and virtual classes/functions 8
4. Testing 9
5. Work log 10
5.1. Week 44 (30th Oct. - 5th Nov.) 10
5.2. Week 45 (6.Nov. - 12.Nov.) 10

5.3. Week 46 (13th Nov. - 19th Nov.) 10

5.4. Week 47 (20th Nov. - 26th Nov.) 11

5.5. Week 48 (27th Nov. - 3rd Dec.) 11

5.6. Week 49 (4th Dec. - 10th Dec.) 12

5.7. Week 50 (11th Dec. - 17th Dec.) 13

5.8. Week 51 (18th Dec. - 20th Dec.) 13

6. Known bugs 14

1. Overview

Our software, as the name suggests, is a traffic simulator that models a small city with a map of streets and vehicles that go along the streets.

Each vehicle will have a source building and a destination building in the city map where they are travelling to; Pathfinding is done using Dijkstra's algorithm. Sources and destinations are randomly generated in the city map, and the sources has a exponential distribution for the frequency at which cars will be leaving. The rate can be also modified via user interface.

The roads have traffic lights at intersections (if more than 3 streets intersecting) to help cars avoid collisions.

The user interface allows interactive addition and removal of traffic sources/destinations and streets.

Users can save and load their created maps. Currently the map size is limited to 16x12, and load functions excepts the maps to be the exact size.

2. Software structure

In this section, we will have a detailed look on our software structure.

2.1. Class structure

Our class structure roughly follows the initial architecture described in the project plan. As stated in the project plan, the software uses `*SFML 2.3.2*` graphics (external) library.

Newer versions of SFML will most likely work as well, but we chose 2.3.2 due to the fact that the university's Linux machines have 2.3.2 already installed.

We've made some small changes, and currently our class structure can be figured out from the software architecture (figure 1).

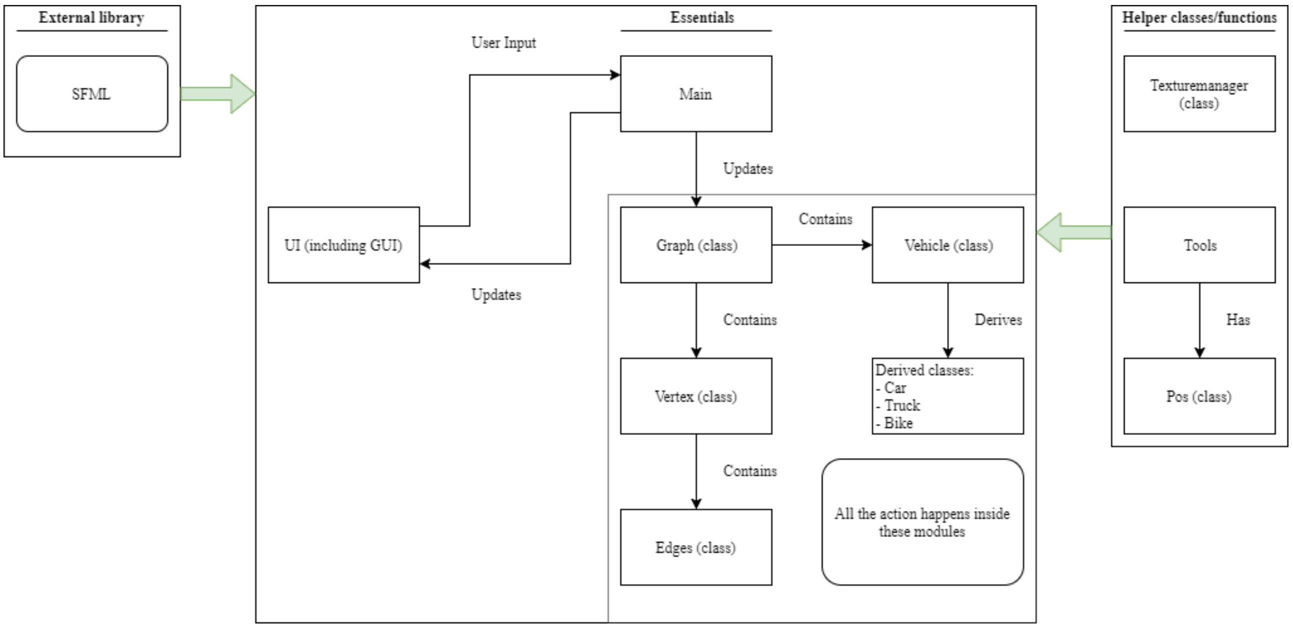

Figure 1: Software architecture of our traffic simulation project "sim-city-1".

2.1.1. main

Main function is responsible for **GUI loop**. It simulates time, handles user input and calls objects to update themselves. In \`*main.cpp*\` file there is also \`*test*\` function for unit testing. Due to scheduling problems, unit testing is not provided in a separate source file.

2.1.2. graph

Graph class stores information about game world. It stores vertices and vehicles and updates them when called. The class also finds paths for cars using **Dijkstra's algorithm**.

2.1.3. vertex

Vertex object represents one tile in game world and a vertex in graph structure. Vertices are linked together by edges and represent one tile type (tileType) so it can be drawn to screen. Vertex also stores information about traffic lights and updates them with help from tools.

2.1.4. edge

Edge objects represent relations between vertices. Each edge is set between two vertices, and it also have weight.

2.1.5. vehicle

Each Vehicle object stores size, position, destination and route of single vehicle. Vehicle is moved when \`*move*\` method is called.

2.1.6. texturemanager

TextureManager class loads textures for tile types (grass, road, building) and returns them when called.

2.1.7. tools

In \`*tools.hpp*\` file there are some additional enumeration and class Pos to easily store and compare coordinates elsewhere in the program.

3. Instructions for building and using the software

To run Sim-city-1, simply navigate to the project root directory using your unix terminal and then run `` `*make run-main*` `` command. This command will compile the source files and make an executable called **sim-city-1**. The command will also run the executable. The executable can be found in the src directory. The program can be run again from there or you can simply use the `` `*make run-main*` `` command again. You can clean the src directory using the `` `make clean` `` command, which will remove object files and executables from the src directory.

To use our software, you should have SFML library installed on your machine, preferably `` `*SFML 2.3.2*` ``. For your convenience, please use Linux systems found in Maarintalo, in classrooms Maari-A and Maari-C, for example.

3.1. How to use the software

Once the software is running, the console will print out a welcome message. You can press **H** to have all available commands printed out to the console.

To add tiles on map, first choose which kind of tile you want to add using keyboard. Pressing **R** adds roads, **B** buildings and **G** grass (empty tile). When you have map ready and want cars to start spawning press **V**.

Press *letter* **O** to change the rate that automatic traffic light control changes lights and **I** to change the rate that buildings spawn vehicles.

Game speed can be changed using numbers **1** (1x), **2** (4x) and **3** (8x). To pause, press **P**, and to unpause, simply set the new game speed.

To save your current map press **S** and follow instructions on the console. To load press **L** and also follow instructions on the console.

It is important to know that when typing in input, be careful with not attempting to type while the console is NOT selected, as key inputs might be registered to the GUI loop, prompting other commands and risking crashing the software. Again, due to scheduling issues, polishing the GUI loop will be done in later versions, if and when it is done.

3.2. Use of C++ features

In this section, we will quickly go through of use of C++ features that were required in the project.

3.2.1. Containers

For containers, we used mainly `` `*std::vector*` `` standard containers. Vectors provide random access, which is very useful, and our code does not require too much memory. Container `` `*std::list*` `` was also used in Graph when storing vehicles because random access was not required for them. In addition to vector and list, we used `` `*std::priority_queue*` `` for Dijkstra's algorithm for vehicle pathfinding. Priority queue is required to get Dijkstra's algorithm to work slightly more efficient than using other methods.

3.2.2. Smart Pointers

We use smart pointers (`` `*std::shared_ptr*` ``) for storing vehicles as our graph class stores vehicles. With shared pointers, we don't have to manually manage memory, which is notorious in C++.

3.2.3. Exception handling

Exception handling is used when user wants to change the rates that 1) change lights and 2) buildings spawn vehicles, as both require user input. The try catch loop ensures that if user enters invalid argument or invalid value, the software throws an error, forcing to retry, or type -1 to cancel the operation. Exception handling is also used when the user wants to load a map.

3.2.4. Rule of three

Rule of three (and rule of zero) is enforced in every part of our software, most notably in our vehicle class. For example, vehicle has a (virtual) destructor, which leads to it also having copy constructor and copy assignment.

3.2.5. Dynamic binding and virtual classes/functions

Vehicle class uses dynamic binding, as we have derived classes for vehicles. For example, virtual destructor is required for base class Vehicle.

4. Testing

Due to scheduling problems, unit testing was done using `<cassert>` and in a separate test function in main.cpp, which can be called while running the main program (instructions can be found in main.cpp). Unit testing was used to ensure that most of the most important individual functions work appropriately.

Unfortunately, there was no time to set third-party unit testing C++ frameworks such as google test or CPPunit. However, as our software evolved from the beginning to the end, debugging was very intensive as the codes was being developed.

To verify that our code works properly, in addition to unit testing, Valgrind was used to check for memory leaks. We concluded that only SFML parts are leaking memory, and only 10 bytes in 1 block was definitely lost – the rest are still reachable. Below is the latest Valgrind with `--leak-check=full`.

```
==18565== HEAP SUMMARY:
==18565==     in use at exit: 143,697 bytes in 1,309 blocks
==18565==   total heap usage: 1,751 allocs, 442 frees, 334,295 bytes allocated
==18565==
==18565== 10 bytes in 1 blocks are definitely lost in loss record 63 of 216
==18565==    at 0x4C2DB8F: malloc (in /usr/lib/valgrind/vgpreload_memcheck-amd64-linux.so)
==18565==    by 0x50C9489: strdup (strdup.c:42)
==18565==    by 0x418328: xstrdup (in /usr/bin/make)
==18565==    by 0x423FC6: target_environment (in /usr/bin/make)
==18565==    by 0x4148D4: ??? (in /usr/bin/make)
==18565==    by 0x4152B3: ??? (in /usr/bin/make)
==18565==    by 0x415BD2: new_job (in /usr/bin/make)
==18565==    by 0x420D9E: ??? (in /usr/bin/make)
==18565==    by 0x4210AE: update_goal_chain (in /usr/bin/make)
==18565==    by 0x407780: main (in /usr/bin/make)
==18565==
==18565== LEAK SUMMARY:
==18565==    definitely lost: 10 bytes in 1 blocks
==18565==    indirectly lost: 0 bytes in 0 blocks
==18565==      possibly lost: 0 bytes in 0 blocks
==18565==    still reachable: 143,687 bytes in 1,308 blocks
==18565==         suppressed: 0 bytes in 0 blocks
==18565== Reachable blocks (those to which a pointer was found) are not shown.
==18565== To see them, rerun with: --leak-check=full --show-leak-kinds=all
==18565==
==18565== For counts of detected and suppressed errors, rerun with: -v
==18565== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

5. Work log

Work log is updated until 15th December 2017 due to the project deadline. Below you can find the detailed description of division of work and everyone's responsibilities.

In addition, there is a description (for each week) of what was done and roughly how many hours were used, for each project member.

In general, every single one of us were busy throughout the entire II period, and workload might be slightly biased towards the end of the course. This can be confirmed with how the git commits are distributed throughout the project development, in II period.

All in all, we are all satisfied with our efforts as a group. Despite the week 50 panic, we were able to pull this off together!

5.1. Week 44 (30th Oct. - 5th Nov.)

Project kick-off

- [ALL MEMBERS]: At least 3 hours spent on internal project kick-off meeting, where we discussed and picked the project topic.

5.2. Week 45 (6.Nov. - 12.Nov.)

Project plan

- [ALL MEMBERS]: At least 3 hours spent on working with the project plan

5.3. Week 46 (13th Nov. - 19th Nov.)

Finalizing Project plan, initial git commits, working with makefiles

- [ALL MEMBERS]: At least 3 hours spent on finalizing the project plan and setting up the project in Maari. Setting up the make file proved to be most laborious.
- [Tianzhong]: At least 2 hours spent on working with first versions of graph, vertex and edge.
- [Risto]: At least 2 hours spent on making a makefile and hello world program.
- [Ilari]: About least 2 hours spent on making makefile.

5.4. Week 47 (20th Nov. - 26th Nov.)

Working on peer review. For our own project: creating essential classes (graph, vertex, edge and vehicle) & main.cpp and learning to work with SFML-graphics to draw something on screen.

- [ALL MEMBERS]: At least 3 hours spent on working together in Maari and doing midterm peer review.
- [Tianzhong]: At least 5 hours spent on working with first versions of graph, vertex and edge, and then with vehicle.
- [Harti]: About 5 hours spent on creating initial versions of edges and vehicles and moving the weights from vertices to edges and creating the data structure for vertices.
- [Risto]: 10 hours spent: 4 hours on researching SFML and creating a GUI loop. 6 hours on dynamic adding and removing of vertices.
- [Ilari]: About 5 hours spent on researching SFML and working with graphics.

5.5. Week 48 (27th Nov. - 3rd Dec.)

Added separate files for vertex and edge (before vertex and edge were declared and implemented in graph.hpp & graph.cpp); Modified the graph so that roads, when added, connect to each other with edges.

- [ALL MEMBERS]: At least 4 hours spent on working together in Maari.
- [Tianzhong]: At least 2 hours spent on issue #7 (const correctness problem).
- [Harti]: About 4 hours spent on modifying the vertices data structure and automatizing the creation of edges between vertices (buildings and roads).
- [Risto]: 6 hours spent: 3 hours on adding a tools.hpp file and moving vehicles on the graph. 3 hours on adding edges on vertices.
- [Ilari]: 5 hours spent working with managing textures and drawing logic.

5.6. Week 49 (4th Dec. - 10th Dec.)

Implementing Dijkstra's algorithm, fixing segmentation fault (e.g. when removing edges), making buildings to create cars automatically, implemented traffic lights and implemented vehicle track their own direction when moving.

- [Tianzhong]: At least 10 hours spent working on texture files and Dijkstra's algorithm. At least 90% of the time was used at working on Dijkstra's algorithm. While the algorithm itself was relatively easy to implement and could've taken less than a few hours, a lot more time was spent due to the inexperience with C++.
- [Harti]: About 10 hours spent on making map modification more user friendly, improving the updating of the GUI, made vehicles spawn randomly to buildings, modified edge weights to suite Dijkstra's algorithm (cars use roads and only start and finish by driving through buildings), improved the functions of traffic lights and gave vehicles a sense of direction.
- [Risto]: 16,5 hours spent: 4 hours on making the vehicles to move between vertices. 5,5 hours on dynamic adding and removing of edges and traffic lights. 7 hours on making multiple cars move on the graph, fast forward functionality and modifying constructors and destructors on vehicles.

5.7. Week 50 (11th Dec. - 17th Dec.)

Finalized Dijkstra's algorithm, fixed vehicle movement logic, implemented ability to save and load maps, fixed segmentation faults when spawning vehicles, implemented Graph::update() to update their route from source to target and, in addition, fixed Dijkstra's algorithm. In addition to aforementioned, vehicles now turn correctly in corners. Last but not least, added some unit testing.

- [ALL MEMBERS]: At least 10 hours working together at Maari together to finalize the project.
- [Tianzhong]: At least 30 hours spent: at least 7 hours with finalizing Dijkstra's algorithm, 3 hours with fixing (weird) segmentation fault when creating vehicles to the Graph and 4 hours with adding console commands for user input. Time was also spent on working with documentation, unit test and fixing minor bugs, not to mention with constant debugging.
- [Harti]: Maybe 40+ hours of finishing off (more improvements on traffic lights, applying Dijkstra's algorithm to vehicles, improved the movement logic of vehicles, modified vehicles to choose random destinations (buildings), improved graphics to use vehicle specs instead of hard coded sizes). Special note: Used 10 hours on last day to create smooth turns, hopefully the course personnel enjoys them more than I did creating them.
- [Risto]: 2,5 hours spent: 2,5 hours on collision detection
- [Ilari]: About 10 hours spent working with documentation, save/load, refining vehicle movement and fixing bugs.

5.8. Week 51 (18th Dec. - 20th Dec.)

Not much left to do except demonstration and reviewing stuff. Below is (predicted) work & duration.

- [ALL MEMBERS]: At least 4 hours spent on 1) demonstrating the project to responsible teacher and peer groups and 2) doing review of other group members, own project and finally peer project. **The reviewing process will be finished on 20th December at latest.**

6. Known bugs

Unfortunately, we did not have enough time to polish our project to 100% perfection. Some minor bugs might hurt perfectionists' minds, and we apologize for the inconvenience.

- Collision detection is not perfect; Vehicles might overlap each other after exiting from turn.
- When vehicles turn to right, they might overlap a little, depending on whether the vehicle type is Car, Truck or Bike.
- User input for controlling traffic light and spawn rates may be unstable.