Respondent: **Tianzhong Pan** Submitted on: Tuesday, 19 December 2017, 3:39 PM

Mid-term review of C++ project

Name of the project group evaluated

sim-city-1

This is a project self-evaluation

☑ Yes

C1.1: The implementation corresponds to the selected topic and scope. The extent of project is large enough to accommodate work for everyone (2 p)

- Our implementation corresponds to the traffic simulator topic and scope.
    - All requirements (stated in the project description) were fulfilled.
- For work distribution, see C2.1 and C2.3.
- **Subpoints: 2/2**

C1.2: The class structure, information hiding and modularization is appropriate, and it is explained and justified in documentation. The file structure corresponds to the class structure (2 p)

- The documentation provides a software architecture of the simulation. Both the figure and the text explains our class structure, information hiding and modularization.
- The file structure also corresponds to the class structure.
- **Subpoints: 2/2**

C1.3: Use of at least one external library (in addition to C++ standard library). Comment the appropriateness of libraries and their use. (2 p)

- The software uses SFML 2.3.2, as stated in the documentation.
    - 2.3.2 was used as Aalto Linux machines has SFML 2.3.2 installed by default, and we wanted to avoid potential compatibility issues.

- **Subpoints: 2/2**

C2.1: Git is used appropriately (e.g., commits are logical and frequent enough, commit logs are descriptive) (2 p)

- In the beginning, the commits had some undescriptive messages, but as the project progressed the messages became more descriptive as 1) we knew more what we were going to do and 2) we became more familiar with git itself.
- Our commits are logical and frequent. The commit frequency is lower on the first half of the II period than the second half, but this was due to the fact that all of our members were very busy during the whole II period, with the workload leaning on the first half.
- Aside from some inexperience problems like one failed merge on 14th December 2017, we think we've used git appropriately enough.
- **Subpoints: 1,75/2**

C2.2: Make or Cmake (recommended) is used appropriately. The software should build easily using these tools without additional tricks. Nevertheless, instructions for building the project should be provided (1 p)

- Make was used in our project, as the project was simple enough to be built using Make.
    - Using Cmake would've made things a bit more difficult: Our inexperience with Cmake combined with scheduling issues results a no-no situation.
- Instructions are clear, and provided in the root README.md.
- **Subpoints: 1/1**

C2.3: Work is distributed and organised well, everyone has a relevant role that matches his/her skills and contributes project (the distribution of roles needs to be described) (1 p)

- The work is distributed and organized well, and a comprehensive work log was provided in the documentation.

    - In general, everybody were involved in programming, but some individual members took specific, individual tasks. For example:
        - TP implemented the Dijkstra's algorithm
        - Risto was the main responsible of working with SFML, with Ilari actively helping Risto.
- **Subpoints: 1/1**

C2.4: Issue tracker is used appropriately to assign new features and bug fixes (1 p)

- We used issue tracker for assigning new features and fixing bugs.
    - However, when we worked together in Maari, there were no big use for issue tracker during the work. Issue tracker was also used as a to-do / to-be-added list.
- **Subpoints: 1/1**

C2.5: Testing and quality assurance is appropriately done and documented. There should be a systematic method to ensure functionality (unit tests, valgrind for memory safety, separate test software and/or something else.) (1 p)

- Valgrind was used in testing memory leaks, and a log was provided in the documentation.
    - We concluded that very most likely all the leaks are due to the SFML parts, and that our code works as intended.
- In addition, some unit tests were done, and the instructions can be found within the main.cpp, as stated in the documentation.
    - Unfortunately, setting up separate unit testing framework failed due to lack of time.
- **Subpoints: 0,75/1**

C3.1: C++ containers are used appropriately (including appropriate use of iterators), and justified (e.g., why certain type of container over another) (2 p)

- 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.
    - Using list for storing vehicles allows efficient use of iterators, without the risk of causing **segmentation fault**.
- In addition to vector and list, we used **std::priority_queue** for Dijkstra's algorithm for vehicle path-finding. Priority queue is required to get Dijkstra's algorithm to work slightly more efficient than using other methods.
- **Subpoints: 2/2**

C3.2: Smart pointers are used in memory management, describe how (1 p)

- We use smart pointers (std::shared_ptr) for storing vehicles as our graph class stores vehicles.
- In addition, Edges are constructed using two shared pointer that points to the vertices.
- With shared pointers, we don't have to manually manage memory, which is notorious in C++.
- **Subpoints: 1/1**

C3.3: C++ exception handling is used appropriately, describe how (1 p)

- Basic exception handling is used when saving or loading a map.
- Better version of exception handling (using try-catch blocks) is implemented for changing the rates of 1) how fast should a building spawn vehicle and 2) how quickly should traffic lights change lights.
    - 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.
- The aforementioned functions require user input, hence the exception handling.
    - Current implementation might cause some bugs, because while the software asks for user input, the GUI loop freezes.
    - Segmentation fault might also occur when changing the spawn rate and lights.

        - Then again, in the demonstration, no crashes occurred.

- **Subpoints: 1/1**

C3.4: Rule of three / rule of five is followed, describe how (1 p)

- 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.
- **Subpoints: 1/1**

C3.5: Dynamic binding and virtual classes/functions are used, describe how (1 p)

- Vehicle class uses dynamic binding, as we have derived classes for vehicles.
    - For example, virtual destructor is required for base class Vehicle.
- **Subpoints: 1/1**

Other comments and feedback to the evaluated project group.

- Despite the panic in week 50, we managed to pull this off.
- In next versions, we should discard the use of console and do everything in the GUI loop. If we are going to continue with our project, that is.
- **Overall score: 5,75+5,75+6=17,5 / 18.**

If you did this review together with (some of) your group members, list the names of the group members here. Everyone needs to turn in a review, either separately or as a group.

Reviewed by all members of group “sim-city-1”.