ELEC-7151 - Object oriented programming with C++

Project Documentation:
Tower Defense 1

Antti Haavikko, 714066, antti.haavikko@aalto.fi
Perttu Jalovaara, 714396, perttu.jalovaara@aalto.fi
Anna Huttunen, 711881, anna.a.huttunen@aalto.fi
Atte Tommmiska, 666716, atte.tommiska@aalto.fi

Turned in: December 11, 2021

1 Overview

1.1 General project description

Our group's project is a tower defense game. The theme of our game is humoristic and close to our lives. In the game, there are geese defending their surroundings from teekkaris, who are trying to wander past the geese. In other words, the towers in our game are geese. There are five different kinds of geese, that have different defense abilities. For example, a `PooperGoose` is able to slow teekkaris down, but it cannot hurt nor kill them. Geese also either have a certain radius of effect or they may choose their victim. Teekkaris may be able to walk through the field with different velocities and using different routes. Teekkaris are not able to hurt the geese, but if too many teekkaris make it through the geese's defenses, they will cause damage and make the player lose health points. This leads to the geese and the player losing the game. The ultimate goal is for the player to survive as many waves of teekkaris as they can.

There are five ready-made levels with different path layouts for the teekkaris. The difficulty of the game increases between each wave of enemies. In addition to the five levels, it is possible for the player to upload their own level to play by using the level editor where the player can design their own level.

1.2 What features were not implemented?

We managed to implement all the basic features listed on A+ subsection Project topics and descriptions. However, we did not implement all the additional features that were listed on the A+ website. Below is a list of the additional features that were left out of the final project.

A list of the additional features we did not implement

- A list of high scores that is saved locally per map, with a username (decide yourself how to calculate points) (1 point)
- Towers can be damaged by enemies (2 points)
- Multiplayer: game can be played from a "client" and a "server side" stores the scores to one place (3 points)

1.3 What features and functionalities were implemented?

When the application starts, the player can choose a level to play from the main menu. Each of the five levels has a unique map. The player has some money in the beginning of the game to purchase new towers (geese). When the player presses on the start wave button, waves of enemies (teekkaris) run through the map following single non-branched paths on their journey by

default, but some enemies choose their path with some intelligence as they calculate the "optimal" path. Whether the paths branch or not depends on the level. Towers can be purchased and deleted during enemy waves. The towers can attack enemies inside their range in different ways and have different methods for acquiring targets. The game is lost when enough enemies reach the end of the path and the player's health points bar (HP) drops down to zero. The player gains money by destroying enemies. Different enemies are worth different amounts of money. Money is needed to build towers.

There are five different types of geese. The basic type is `BasicGoose`, which causes a low amount of damage to enemies and shoots with a slow frequency at enemies inside its range. The second type is `PooperGoose` which slows down enemies within its range. The third type is ShotgunGoose which shoots a flock of bullets at the nearest enemy. The fourth type is `SniperGoose`, which chooses it's target based on the remaining path length of the enemy. `SniperGoose` causes significant damage with a special golden bullet. The fifth type is `MamaGoose`, which shoots plasma balls with a great frequency. Each type of goose can be upgraded which increases the range. Increasing the tower radius by upgrading a tower costs the player 50 dollars. Different types of towers cost different amounts of money. Figure 1 showcases concept art for `SniperGoose` and `MamaGoose`.

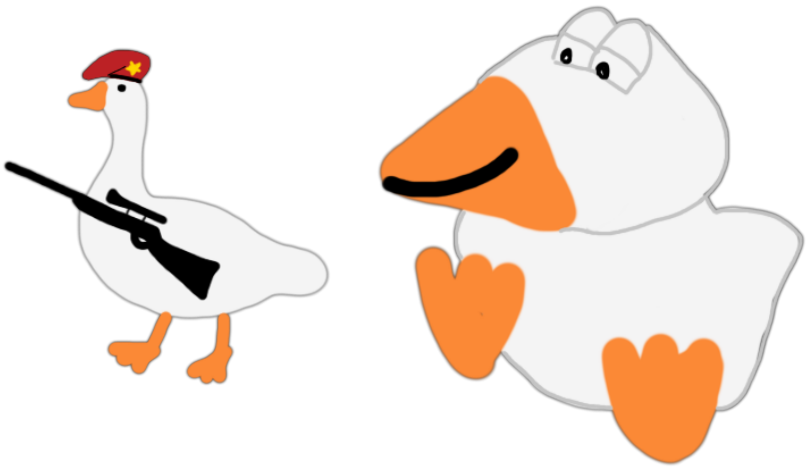

Figure 1: Illustrations of `SniperGoose` (left) and `MamaGoose` (right).

There are also five different types of enemies. The first type is `Koneteekkari` which chooses its path randomly. The second type is `Fyysikko` (Physicist), which chooses the most optimal, that is the shortest, path to cross the game field, but is relatively easy to destroy. The placement of towers changes the optimal path for the third type, which is `Kylteri` (Biz student). A `Kylteri` is a bit harder to kill but the reward for killing is significantly higher than for the earlier types. The fourth type is a `Cruiseship` which takes many hits to

kill and once it has been destroyed it splits into other enemies listed above. The fifth type is `Dokaani` which always chooses the longest path there is. The effect of a tower on an enemy depends on their types. The `PooperGoose` for example is really effective at slowing down most enemies, but it can not slow down a `Cruiseship`.

The game is controlled by the player's cursor. The user can click on a tower icon to purchase it and then drag and drop the tower on the map. The player can also upgrade and delete the towers by first clicking on the tower and then selecting either upgrade or delete. The number of enemy waves, enemies that have crossed the field and the amount of money are displayed on the user interface at all times. Once the game ends in defender failure, the player is shown the game over screen along with the enemy wave number they reached. The game has sound effects for enemy destruction, tower purchasing, player losing hp and for the game over screen.

A list of the additional features we implemented

- Non-hardcoded maps, i.e. read from a file or randomly generated
- Upgradeable towers
- More different kinds of enemies and towers
- Level editor to create levels and to save them in a file
- New towers can be purchased during an enemy wave
- Multiple paths of the enemies, so branched paths and enemies choose one with some intelligence
- Dynamic enemy paths that are altered with the placement of towers
- Sound effects
- Some attack types may be more effective against some defense types. For example, `PooperGoose` cannot slow down the `Cruiseship`.
- Main menu with nice visuals
- Different towers have different ways to acquire targets
- A healthbar for the player hp
- Clear all towers button for easier testing (or for hardcore players)

2 Software structure

Our tower defense game is written according to the object-oriented programming paradigm, as required by the project guidelines. In this section, we introduce and discuss the class hierarchy and the overall architecture of the software. The tower defense genre allows for a very intuitive approach to defining classes for the game. Our software consists of 20 custom classes, of which two are abstract and one is templated. In addition to our custom classes, we utilize classes from the Qt graphics library, which acts as the backbone of our software. Figure 2 presents the class relationship diagram of the software. Arrows are used in the diagram to represent inheritance relationships. The inclusion hierarchy is roughly outlined by the positions of the classes, e.g., `Game` includes `BuildIcon`, `Tower`, `Enemy`, etc.

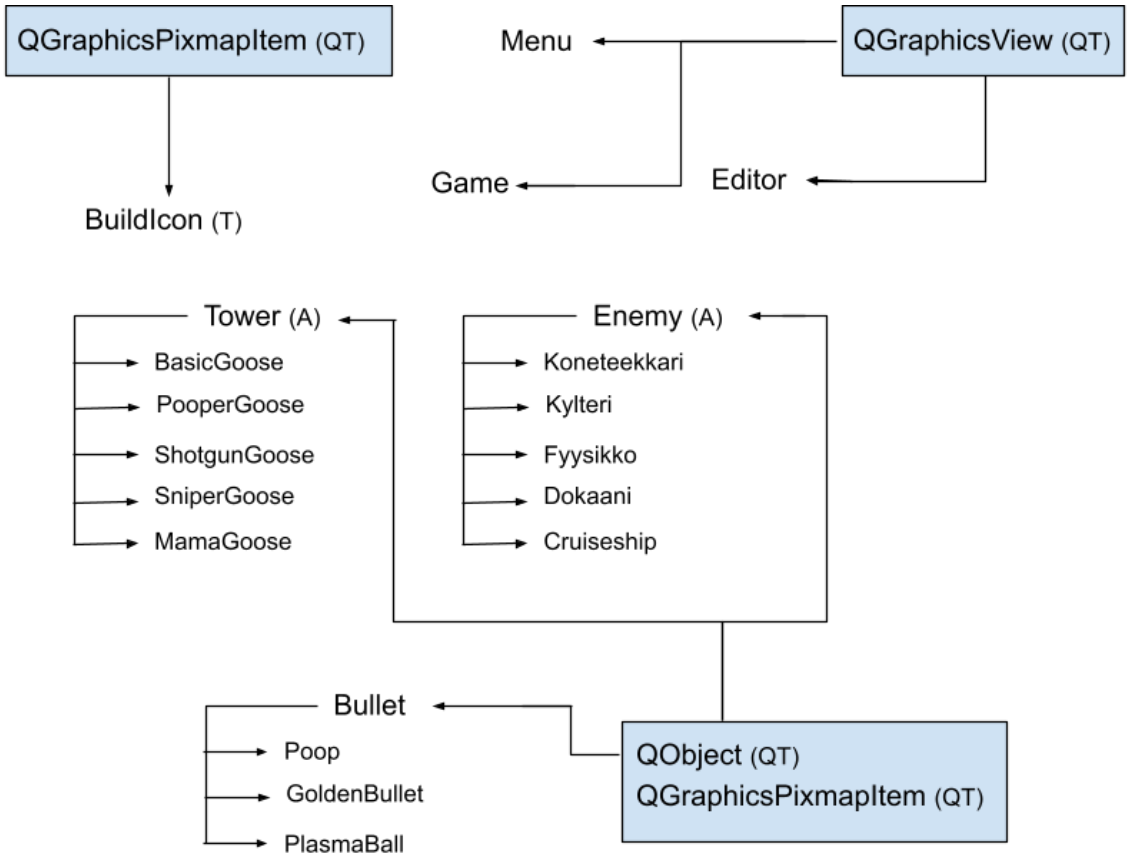

Figure 2: The class relationship diagram of our game. The arrows represent inheritance relationships. An abstract class is denoted with (A) and a template class with (T). The classes from Qt library are marked with (QT).

Before diving too deep into the class hierarchy of the software, we must clarify and justify a couple of our architectural decisions. First, the lack of adherence to the rule of three (and the rule of five) must be addressed. The classes `Editor`, `Game`, `Enemy` and `Tower` have explicit destructors because of dynamic memory allocation. Yet, none of these classes have explicit copy constructors and copy assignment operators (nor move constructors and move assignment operators). Clearly, this violates the rule of three (five). We justify this decision rather naively: We do not copy or move any instances of the aforementioned four classes in our code and thus the (incorrect) implicit constructor and operator definitions are never used. Naturally, if our tower defense game were to be expanded, it would be wise to implement the corresponding explicit definitions to prevent future problems.

The second set of design decisions that we want to elaborate further concerns the graphical user interface of our tower defense game. Instead of opting for the standard (or at least more common), single-window GUI, we decided to develop a multi-window experience. There are pros and cons to both approaches, but having the software run in multiple windows seemed like the premiere option for us. Using the Qt library, we made the choice of having each game instance have its own window. This is implemented with the `QGraphicsView` class, which can act as a separate window. The possibility of running multiple games side-by-side increases the skill cap of Goose TD significantly. An avid speedrunner could, for example, play all five levels at the same time and thus display a higher skill level in comparison to the single-window approach. All active windows, i.e., games and level editors are initially opened from the main menu. This brings us back to the class hierarchy of our software, starting from the `Menu` class.

`Menu` is the top-level class of our software. It inherits the `QGraphicsView` class and is (transitively) responsible for the memory management of all object instances. In general, the Qt library relies on parent-child relationships for freeing memory, i.e., when a parent is deleted, all of its children are also deleted. Thus, in our code, most objects receive a parent parameter during construction. The only exceptions to this are certain graphical objects that inherit the `QGraphicsPixMapItem` class, because they are manually added to a `QGraphicsScene` that acts as a parent substitute (foster parent?). The `Menu`, `Game` and `Editor` classes all have their own scene which is cleared, i.e., all objects added to the scene are deleted, upon destruction.

In addition to providing the graphical elements and simplifying memory management, the Qt library has a feature called signals and slots for inter-object communication. The signals and slots are enabled by calling the `Q_OBJECT` macro within the header file of a class. The principle behind signals and slots is rather simple: objects can emit signals that will then activate the connected slot, i.e., a class method that responds with a wanted action.

Slots and signals can be connected inside a single object or between different objects. We use the signals and slots in combination with the `QTimer` class to implement the main gameplay loops, such as movement of `Enemy` and `Bullet` objects, `Tower` target acquisition and attacking as well as spawning waves of enemies. All of the buttons on our GUI are also reliant the signals and slots feature. Most of these buttons are `QPushButton` objects, which proved to be effective in adding various functionalities to our software.

As indicated in Figure 2, `Tower` and `Enemy` are abstract classes that are inherited by every type of tower and enemy. Abstracting the default parts of these classes facilitates the future extension of the software with additional types of tower and enemies. Adding a new type of enemy is simple due to the inheritance structure and virtual functions. It only requires the definition of a custom constructor which initializes the *stats* (health points, price, damage, speed) of the enemy and a `ChoosePath` function which specifies how the enemy chooses which path it is going to take. Defining new tower subclasses is done similarly, but one needs to define two functions in addition to the constructor (`AttackTarget` and `AcquireTarget`). In a similar fashion, the `Bullet` class is inherited by its subclasses. However, unlike `Tower` and `Enemy`, the `Bullet` class is not abstract as it can be used to create basic bullets, which are used by the `AttackTarget` methods of `BasicGoose` and `ShotgunGoose`.

Instead of using `QPushButton` objects to implement the graphical elements for purchasing towers, we decided to define a custom template class, namely `BuildIcon`. This gave us more control over the functionality and form of the build icon. Each `BuildIcon` object receives a class parameter `T`, which corresponds to the `Tower` subclass that we want to build with the icon. After researching template classes, it became clear that the declaration and definition of a template class cannot simply be separated into a header and a source file. Thus, unlike all other classes in our software, `BuildIcon` is declared and defined in just the header. To accomplish this, we used inline function definitions for the constructor and the `mousePressEvent` method.

The aim of this section is not to elaborate every line of code in our software. However, one more structural feature deserves highlighting: the serialization of levels. By using the level editor, the user can create custom levels, which are converted into serialized binary data files. This custom serialization format can be read with the `ReadPathsFromFile` method of the `Menu` class. The implementation of this serialization uses the `QDataStream` class, which is capable of writing Qt types, such as `qint32`, into binary. For more specific information about the classes and their member functions see the Doxygen documentation in the `doc` directory. Naturally, the source code can also be looked at if the documentation does not suffice.

3 Instructions for building and using the software

3.1 How to compile the program?

Our program uses the Qt graphics library. Make sure you have this library downloaded and installed on your computer. To link and compile the program we first use CMake, which creates a `Makefile`. Only one line needs to be changed in the ready-made `CMakeLists.txt` file. Then we compile with Make by running the `make` command. We get an executable that we can run, that is, the actual game. Below are the step-by-step instructions on how to compile and get the program running on Unix based operating systems. If you are using Windows Subsystem for Linux (WSL), you need to additionally set up a server for the graphical user interface (see, e.g., VcXsrv) in order to play the game due to the lack of native GUI support in WSL.

1. Open Terminal (MacOS) or Command Line (Linux).
2. Using your favourite package manager download the Qt library.
   `> brew install qt@5` (example)
3. In terminal, navigate to the tower defense directory.
   `> cd path/to/the/tower-defense-1`
4. Open `CMakeLists.txt` in your favourite text editor.
   `> nano CMakeLists.txt` (example)
5. On line 22, change the `CMAKE_PREFIX_PATH` to match the Qt's `cmake` directory which includes the configuration files (e.g. `Qt5Config.cmake`).
   `> set(CMAKE_PREFIX_PATH path/to/Qt/lib/cmake)`
6. Navigate to the `build` directory.
   `> cd build`
7. Run CMake which uses the instructions in the text file `CMakeLists.txt`.
   `> cmake ..`
8. The previous step should have created a `Makefile` in the `build` directory so run make to create the object files and an executable.
   `> make`
9. Open the game by running the executable.
   `> ./game`

The instructions for CMake and for the changes in the `CMakeLists.txt` can also be found in the `readme.md` file of the `build` directory.

Error handling
In case of errors with the compilation, we recommend having a good look at the error message and using Google as your search engine.

3.2 How to use the software?

After compiling the program according to the previous subsections instructions, the graphic user interface (GUI) should open. The main menu opens and it looks like the menu in figure 3. The illustrations introduce the topic of the game and the logo tells the name of the game. The GUI responds to the commands given with a cursor. The player chooses a level from the default levels 1 to 5 or, alternatively, chooses a custom level. If the player clicks CUSTOM LEVEL, but there is no custom level created, a text appears advising the player to "create a custom level in level editor first".

Figure 3: The menu that opens as the game is opened.

Clicking on the LEVEL EDITOR opens a window, that has two buttons on top. By clicking TOGGLE PATH CREATION the player can create the points of a new path by clicking anywhere on the window. By clicking the TOGGLE PATH CREATION button again, that path is finished. Another path can be created by clicking TOGGLE PATH CREATION once again. Once the paths are finished, clicking SAVE TO FILE saves the level. Now, when CUSTOM LEVEL is clicked in the menu, the level that was just created opens.

When the player has chosen the level they wish to play, a new window containing that level opens. Figure 4 shows an example of what that window may look like. In the upper left corner, all possible towers are shown with their prices. The towers can be dragged and dropped on the game field, if the player has enough money for them. The towers automatically attack the enemies in their range. Each tower's range can be upgraded by clicking the button UPGRADE next to them, or a tower can be deleted by clicking DELETE. All towers can be deleted simultaneously by clicking CLEAR TOWERS in the lower left corner. The *real* gameplay starts when START WAVE is clicked and an enemy wave starts moving across the map following allowing paths.

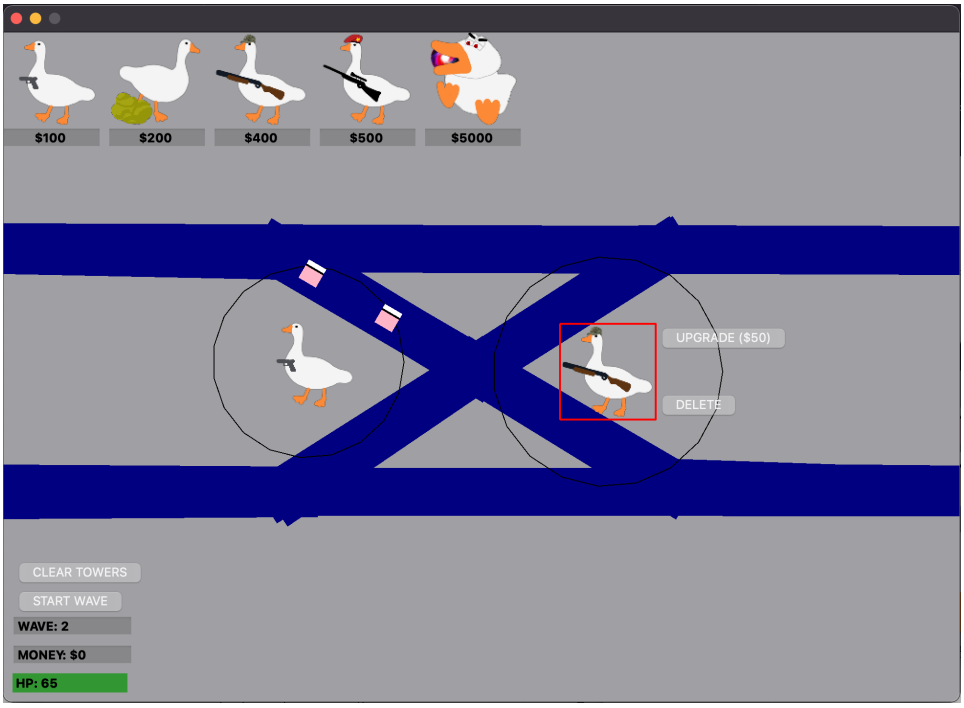

Figure 4: An example of a level in the game.

The lower left corner contains three bars showing progress in the game. The money attribute of the player increases as enemies are destroyed and shrinks as towers are bought or upgraded. The player starts off with the health bar in maximum 100 health points (HP). The health bar decreases the more enemies have crossed the entire map. If all health points are lost, i.e. enough teekkaris have managed to sneak through, the game is over.

The WAVE counter shows how many enemy waves have been started by clicking START WAVE. The waves get more and more challenging and when the player eventually loses the game, they are displayed how many waves they survived. The player may exit the game via the exit button at any time. Another level can be chosen and it opens another window.

4 Testing

As our project is a game that is used via a GUI, most of our testing revolved around implementing new features and testing them using the GUI. Most of our rigorous testing revolved around checking the program for possible memory leaks, and checking that the program works as intended, when we have multiple games running and so on. For example, we manually started a few games from the main menu, closed one of the games from its own window, and then closed the main menu window. After that we checked via valgrind if any memory was leaked.

In our `tests/` directory we have three different tests. The first test (Figure 5) opens a `Game` window so that the tester can ignore the main menu. The second test does the same for the `Editor`. The third test opens and closes a game by itself. It adds two different towers, removes them using the game's `ClearTowers()` method, adds two new towers, starts a new wave, and then closes the window. The purpose of this test is to check for memory leaks within the game.

Most of the problems that we had during testing were related to tower removal and wrong parent hierarchies within the program. That is sometimes we would try to remove an object twice, not realizing that the library does the removal automatically.

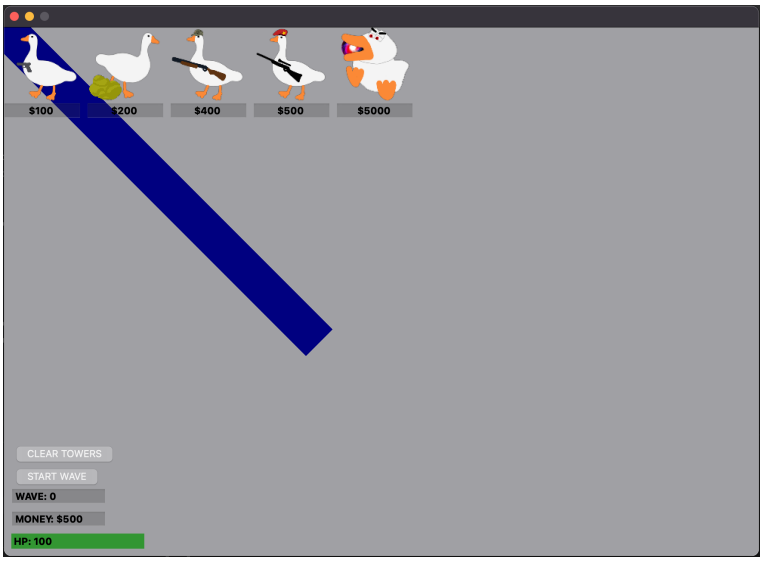

Figure 5: The `Game` window of the first test.

5 Work log

Here is a description of the division of work and everyone's responsibilities. The responsibilities and work load were distributed based on everyone's interests and schedules. The rough weekly hours of each member are included in table 1. A summary of work done by each member is included in the end.

5.1 Division of work and responsibilities

Every group member will naturally participate in writing code. Additional duties and responsibilities are listed below.

Antti's responsibilites:

- Project Overseer
- Writing the Makefiles (CMake)
- Updating the `README.md` files (Markdown)

Perttu's responsibilites:

- Project Overseer
- Writing and updating the `Meeting-notes.md` (Markdown)
- Memory management

Anna's responsibilites:

- Project Plan
- Object diagram (layout of main classes and modules)
- Graphical design

Atte's responsibilites:

- Final source code documentation (Doxygen)
- Coding style
- Testing
- The use of Git

Table 1: Hours used on the project by each member per week.

Member \Week	44	45	46	47	48	49	Total
Antti	15	10	8	12	15	20	90
Perttu	10	5	5	10	20	25	75
Atte	6	4	4	8	10	15	47
Anna	4	2	0	6	2	3	17

5.2 Summary of work

Antti's work:

- Studied the A+ project materials.
- Tested CMake in combination with the Qt library.
- Wrote the `CMakeLists.txt` file
- Implemented a small graphical test as a demo.
- Coded the demo version of the game according to the *C++ Qt Game Tutorial* by Abdullah.
- Basic towers and bullets.
- Start round and clear towers functionality.
- Healthbar. The money earning principle. Cruiseship enemy.
- Dynamic path selection and target acquirement with Perttu.
- Made upgrading towers possible.
- Sound effects with Perttu.
- Some fixes to the memory handling with Perttu.
- Code documentation with the help of Atte.

Perttu's work:

- Studied the A+ project materials and Google's C++ Style Guide.
- Brainstormed game ideas and came up with Goose TD.
- Sketched concept art for geese and finished the Project Plan.
- Dynamic path selection and target acquirement with Antti.
- Acted as an intelligent rubber duck for Antti during coding.
- Centered all graphical elements.
- Fixed aiming by learning about the coordinate system of Qt.
- Made the level editor along with level serialization.
- Sound effects with Antti.
- Most fixes to memory handling.
- Code documentation with the help of Atte.
- Designed the final main menu GUI and the game logo.

Anna's work:

- Studied the A+ project materials.
- Did the majority of the work on the Project Plan.
- Brainstormed visuals for the game.
- Drew the enemy graphics.
- Implemented PooperGoose.

Atte's work:

- Studied the A+ project materials.
- Resolved challenges that appeared in compiling the graphical test.
- Fixed goose graphics resources.
- Generated the first version of Docs for the game.
- Main menu.
- Valgrind memory tests.
- Some fixes to the memory handling with Perttu.
- Code documentation (Using Doxygen etc.).
- Made sure that the code conformed to the coding standards of the course.