UNIVERSITY OF BURGUNDY

C++ Project Report

S.Mehdi MOHAIMENIANPOUR
A K Erfanul ASSAD

Bachelor in Computer Vision

May 15, 2015

Abstract

This report presents a computer game for programmers with codename of `Conquer VoidLand` , our project as part of the Computer Science course module in the second semester of study in the Bachelor in Computer Vision program at the University of Burgundy, France. The course objective was to provide the necessary background in Software Engineering and advance programming using `C++` under `Qt.` framework. This game was inspired by Platinum Rift II game from Codingame [1] website, the concept is likewise but the rules and gameplay has changed. The challenges faced and our solutions are described, followed by a discussion of the future work for this project. The source code can be found as an open-source project in the following GitHub repository:
https://github.com/Voidminded/conquerVoidLand

CONTENTS

List of Figures

CHAPTER 1

INTRODUCTION

The objective of this project assignment was to assess the programming ability using `C++` language under `Qt`. framework. This project was inspired by CodingGame[1] Platinum Rift II game and the concept is the same, a game for programmers with the concept of conquering a planet with an AI designed by player. But rules, game play, number of players, platform, and... are changed. In this project tried to use nearly all the aspects of C++ language and object oriented programming which was thought in the Computer Science course and some of the many features of Qt. framework.

This project is being developed under Linux operation system (Fedora 21) and should work on all the Linux machines, it's not cross-platform yet and all the system commands that is being used in child processes such as copying and compiling the players code, running the codes as a child process and interacting with them are Unix commands.

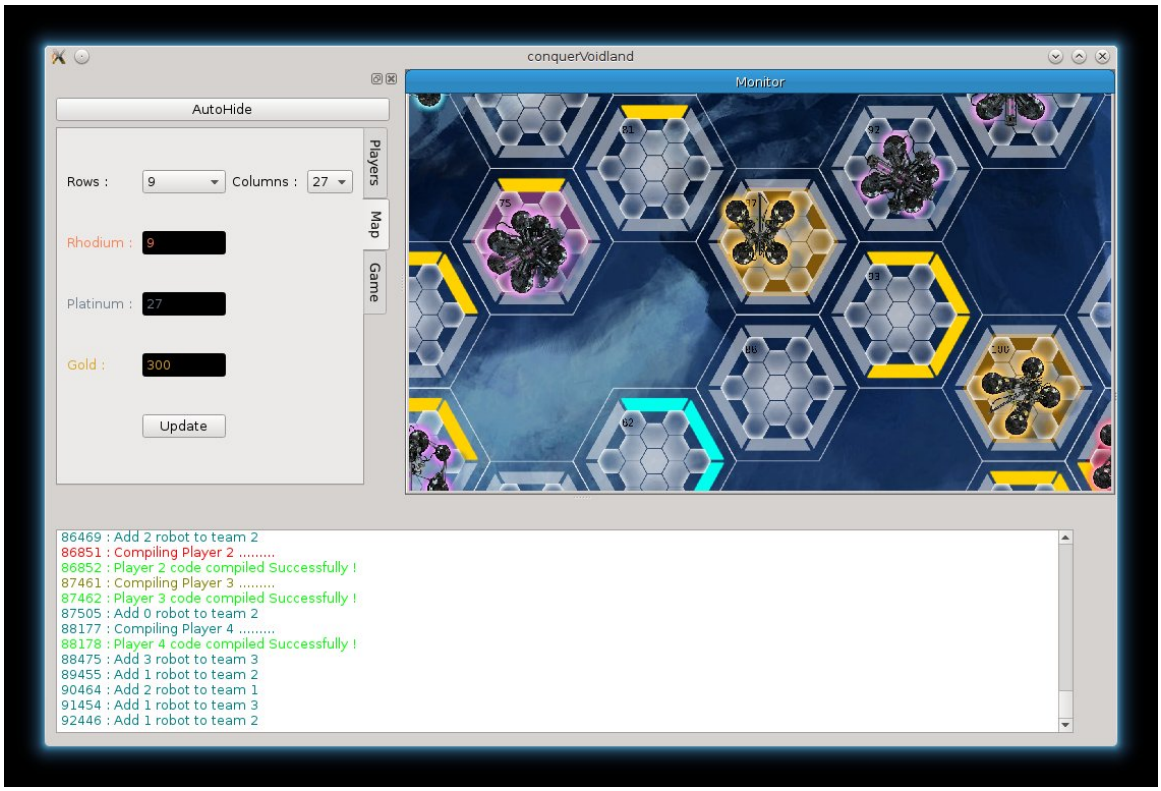

Figure 1.1: Screen shot of the game

The C++ version that this project is being developed by is GNU, C++14[2] compiler, it isn't tested by C++11 and C++0x but it should compile and run without any problem.

The Qt. Framework version used for developing this project is Qt. 5.4[3], the features used from this version (such as QMDiArea instead of QWorkSpace) make the project incompatible with Qt. 4.x and it can not be built with them.

CHAPTER 2

PROGRAM STRUCTURE AND FEATURES

As this project is really huge and we should keep this report as short as possible (You mentioned to keep it less than 20 pages and we *TRY* to), and also we don't want to bore you by going to all the details of classes, in this chapter we explain *shortly* the special classes and their special features.

2.1 BASE.H

This file contains important information that could be used in all the code. Information such as refresh frame rate, communication interval with players, time allowed to players to present a response, some mathematical constants and ...

One of the thing that is being done in this file and is to enumerate the expressions used in code so that we can use them globally.

This part enumerate the color:

```
enum RobotColor{
    Red = 1,
    Yellow = 2,
    Cyan = 3,
    Magenta = 0
};
```

Or enumerating the Robots types:

```
enum RobotModels{
    Explorer = 0,
    Ninja = 1,
    Terminator = 2,
    Predator = 3
};
```

And the materials:

```
enum Material{
    Rhodium = 1,
```

```
    Platinum = 2,
    Gold = 3,
    Neutral= 0
};
```

But the most significant part of this file, is proper using of `Macro` in C++. As a lazy programmer, writing `setter` and `getter` functions for all the variables in classes is a tedious work. The easiest solution that comes in mind is to define everything as `public` easily and we don't need `setter` and `getter` functions, but this will endanger the `Encapsulation` aspect of object oriented programming. So, we came up with another idea, using Macro for generating the `setter` and `getter` functions.

As shown below this macro takes the variable `type` and a `name` for global name which will add a `set` and `get` before that as public getter and setter functions, and a `local` name for variable. It creates the local variable as `protected` so that in inheritance this will be passed to the child class.

```
#define Property(type,name,local) \
        public: inline type& get##name() {return local;} \
        public: inline void set##name(type val) {local = val;} \
        protected: type local
```

One issue that should have in mind for using this is that it will change the scope to `protected` and everything being defined after this will be `protected`, so it's better to be used at the end of the scope. And the following macro acts the same but only creates the `getter` function so that the variable can not be modified from outside of the class.

```
#define PropertyGet(type,name,local) \
        public: inline type& get##name() {return local;} \
        protected: type local
```

2.2 KNOWLEDGE CLASS

This class is the brain of the code. It acts like a blackboard that all the other classes can use it for communication. Any variable or function in this class is being accessible for each file that includes the header of this class `knowledge.h` without needing to create an instance of this class. This could be done by creating an instance of this class at the `main` function and pass the pointer to all the other classes. But the best idea that came in mind was to create one pointer to this class at global scope (done at the top of `"knoeledge.cpp"` file) and in it's own header file and `extern` it to be reachable for all other external usages. Once this is done we can `new` this pointer once and create one instance for usage, which is done in `main.cpp` file.

```
extern CKnowledge* knowledge;
```

2.3 ROBOTS CLASSES

The `CRobot` class contains all the common data in the four kind of robots that we have such as `price`, `hitPoint`, `possition`, `direction`, and etc. to avoid reimplementing these in all the robots classes. In the other word this class is the base of all the robots. It also controls the hovering of the robot, for having a better and more beautiful UI, we designed the robots to have an slight ∞ shape movement while they are standing on a map continent to simulate the hovering and flying a robot. This is done by a `QTimer` for each robot instance that moves the robot by the refresh rate on ∞ shaped path each time it's timer is `timedOut`.

```
connect(shiftTimer
            , SIGNAL(timeout())
            , this
            , SLOT(shifting()));
```

It also contains some `virtual` functions that should be filled by it's children such as:

```
virtual int hit(){return 0;}
```

The other four robot classes `CExplorerRobot`, `CNinjaRobot`, `CTerminatorRobot`, and `CPredatorRobot` inherit the this `CRobot` class and add their own unique members to a robot. One thing that is considered here is defining the `draw` function for these classes as static member function:

```
static QPixmap draw(int color, int dir);
```

Thus for generating the robots images this function if just being called once at the start of program upon creation of `CKnowledge` class instance as mentioned before, and there will be no need to create an object of these classes for calling this function. The only reason this function is placed in these classes is that they should have a copy of all the possible image of their robot in themselves to return in upon request to `CMonitorWidget` to draw them on the scene.

Finally all the headers of all four kinds of robots are included in `robots.h` file, so that any other class who needs to include them can do it by including `robots.h` file(again because of laziness).

2.4 CUSTOMIZED DOCKABLE TAB WIDGET CLASS

The `CTabDockWidget` class is designed to be added to workspace in main window and dock itself to right or left of the screen (defined when calling the constructor of the class) and then you can add any other widget as a new tab to this widget. For instance adding the map generation widget to main window is being done by:

```
mapWidget = new CLoadMapWidget(this);
tabWidget->tabs->addTab(mapWidget, "Map");
```

For this widget the auto hide ability is implemented so that in small screens you can activate the auto hide, it will hide when you don't use it and will reappear as soon as the

mouse cursor goes to the edge of the screen, the speed of hiding and reappearing can be controlled in the widget source code.

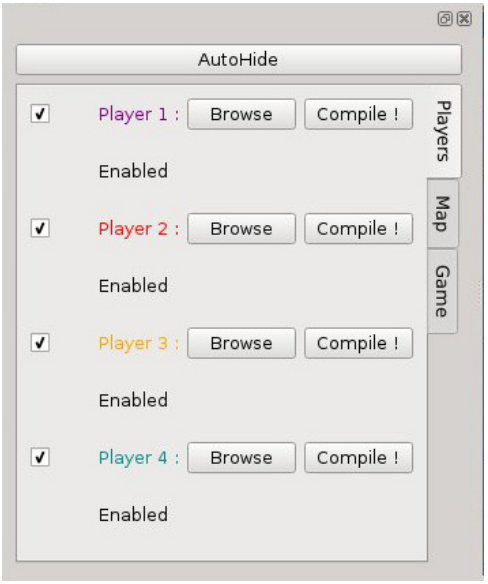

Figure 2.1: Docable Tab Widget

2.5 Status Widget

This widget is being implemented as an internal console in the code, so that we can print everything we need such as game information, players moves, map generation result, and... on it. It contains a `QTextEdit` area that prints everything on this and has the ability to print the text in any color. The text area had Auto-Scroll function and had a buffer that is set to 1000 lines, when this buffer fills everything on the text edit area will be deleted automatically. By using this widget printing everything in code is so easy, as shown bellow:

```
write(QString str, QColor color = QColor("black"));
```

It gets a `QString` and print it by the chosen `QColor`. If the color is not indicated, the text will be printed in black.

Before each line of output on this widget, a number will appear which is the time since program is started in milliseconds, so that we can keep trace of what expactly happen when in the program.

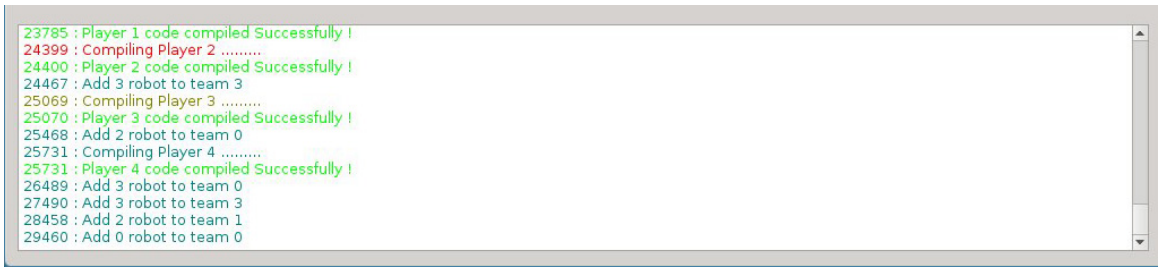

Figure 2.2: Status Widget

2.6 Game Class

The game class has the responsibility to run the game and communicate with players. When you enable a player and select his/her source code in load player widget and click on `Compile` button, the game class will compile the code and show the result. If the program compilation fails this class shows the `gcc` compiler error in red on the status widget, otherwise give a successful compilation message.
When the game starts, this class creates one child process for each of the players, on the exact time defined by `_COMMAND_INTERVAL` in `base.h` file feed each players process with the game input, waits for exactly $100ms$ as defined in game rules for the process to provide output, and continue the game by executing the players commands for moving the robots (if the movement is valid) and purchases (again of course if the purchase is valid). If the player does not provide output on this period, this class will kill the process and kicks the player out of the game.
One way to implement this class was using `QThread`, as multi thread program and one thread for each player, but using `QProcess` as a child process for each player was way easier and in fact we don't need a separated thread for each player as they are some console application and we should monitor exactly the execution time of them.

2.7 Map

Implementing the map for this game was quite a challenge, as the continents in this game are hexagonal saving the working with them was hard.
The first idea that came in mind was implementing a new data structure for saving the map, but this data structure became so complex in designing and using that was even harder and process consuming. The next idea that professor gave us was using a matrix for saving the connection between the continents, which means each row and column of matrix has at most 6 true values which is the connection of this continent with 6 other adjacent continents. But again, having this could not help working with map, imagine we know that continent A is connected to continent B, but for drawing it where should we put them ?! put the continent B above A, under A, upper right, lower right, upper left or lower left ?! so even this idea did not solve our problem.
After some thinking we came up with an idea ! If we save the map in a simple $2x2$ matrix like a square grid, and then shift the odd columns half a cell down, then each cell has 6 adjacent cells as shown in Fig.2.3.

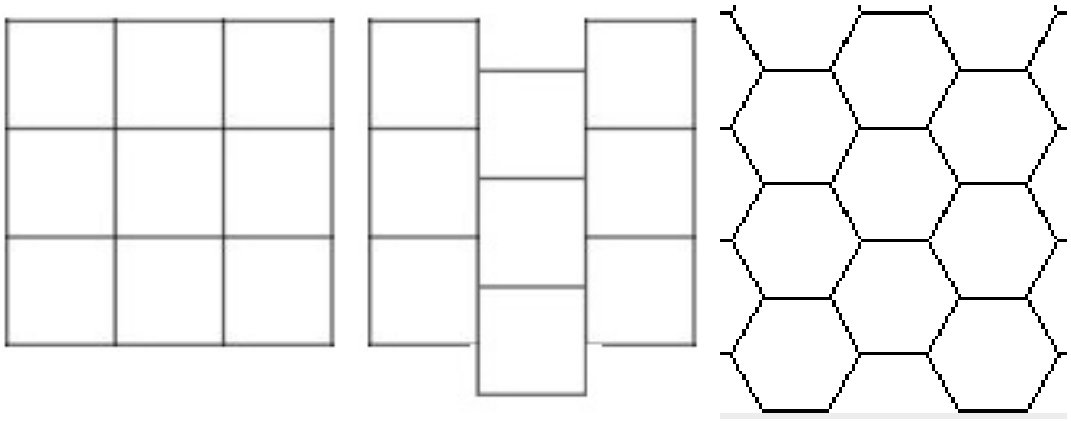

Figure 2.3: Converting a regular grid to hexagonal grid

Of course implementing this idea had it's own challenges such as for each cell detecting if it's column is odd or even:

```
if(activeCells.contains(QPair<int, int>
    (it->first+1, it->second)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first+1,it->second)]));
if(it->first%2 && activeCells.contains(QPair<int, int>
    (it->first+1, it->second+1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first+1,it->second+1)]));
if(!it->first%2 && activeCells.contains(QPair<int, int>
    (it->first+1, it->second-1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first+1,it->second-1)]));
if(activeCells.contains(QPair<int, int>
    (it->first, it->second+1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first,it->second+1)]));
if(activeCells.contains(QPair<int, int>
    (it->first, it->second-1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first,it->second-1)]));
if(activeCells.contains(QPair<int, int>
    (it->first-1, it->second)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first-1,it->second)]));
if(it->first%2 && activeCells.contains
    (QPair<int, int>(it->first-1, it->second+1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first-1,it->second+1)]));
if(!it->first%2 && activeCells.contains(QPair<int, int>
```

```
    (it->first-1, it->second-1)))
    mySet.insert(QPair<int, int>(it.key()
        , forwardPos[QPair<int,int>(it->first-1,it->second-1)]));
}
```

Or detecting the robots direction during moving:

```
if(knowledge->map->reversePos[from].first == knowledge->map->reverseP
{
    if(knowledge->map->reversePos[from].second < knowledge->map->reve
        knowledge->teams[id].explorers.at(j)->setDirection(0);
    else
        knowledge->teams[id].explorers.at(j)->setDirection(3);
}
else if(knowledge->map->reversePos[from].first < knowledge->map->reve
{
    if(knowledge->map->reversePos[from].second == knowledge->map->rev
    {
        if(from%2)
            knowledge->teams[id].explorers.at(j)->setDirection(4);
        else
            knowledge->teams[id].explorers.at(j)->setDirection(5);
    }
    else if(knowledge->map->reversePos[from].second > knowledge->map-
        knowledge->teams[id].explorers.at(j)->setDirection(4);
    else
        knowledge->teams[id].explorers.at(j)->setDirection(5);
}
else
{
    if(knowledge->map->reversePos[from].second == knowledge->map->rev
    {
        if(from%2)
            knowledge->teams[id].explorers.at(j)->setDirection(2);
        else
            knowledge->teams[id].explorers.at(j)->setDirection(1);
    }
    else if(knowledge->map->reversePos[from].second > knowledge->map-
        knowledge->teams[id].explorers.at(j)->setDirection(1);
    else
        knowledge->teams[id].explorers.at(j)->setDirection(2);
}
```

But as this idea seemed to be a genius idea we followed that.

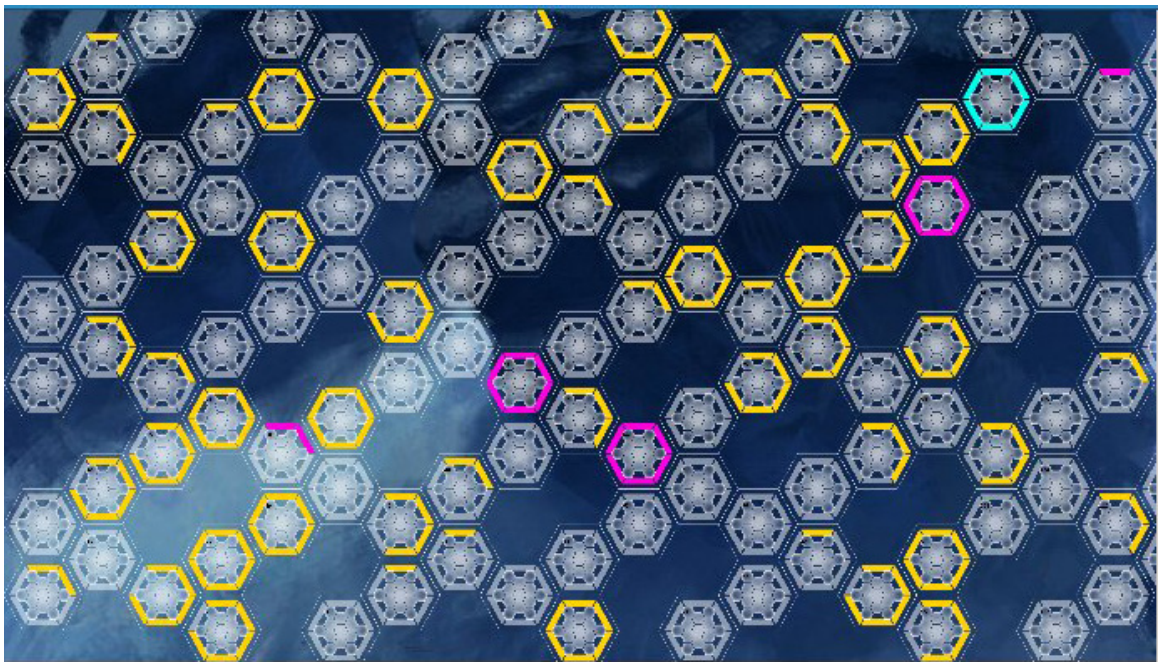

Figure 2.4: The Game Map

2.8 Other Widgets

This program contains also some other widget classes such as:

- `CLoadPlayersWidget` class that is s simple widget with some `QPushPutton`, `QLabel`, and `QFileDialog` for loading the players codes in GUI.
- `CLoadMapWidget` class with some `QComboBox`, `QLabel`, `QLineEdit`, and `QPushButton` for generating a map with given size containing given resources.
- `CGameWidget` which is using for stating and controlling the game sequence.

CHAPTER 3

GRAPHICAL USER INTERFACE

The GUI of this application is completely designed by Qt. Framework features without using Qt. UI form designer and all are coded, one my think that this work is reinventing the wheel ,but I, for one, found that writing the code by hand was conceptually simpler than using Qt Designer, although for complex GUIs Designer might make sense. Large GUIs might be possible using Designer, but with time they might become very difficult to manage as complexity increases. The purpose of not using UI designer and hand coding:

- Control. If you have a layout where you need to instantiate/initialize the controls in a very particular order, or dynamically create the controls based on other criteria (In this project creating and updating some classes instance before some other), this is the easiest way.
- f you have custom widgets, you can kind-of-sort-of use the Designer, adding the closest built-in QWidget from which your class derived and then "upgrading" it. But you won't see a preview of your widget unless you make it a designer plugin in a separate project, which is way too much work for most use cases.
- If you have custom widgets that take parameters in their constructor beyond the optional QWidget parent, Designer can't handle it. You have no choice but to add that control manually.
- Best choice if you need to add or remove widgets at run-time.
- With discipline, you can still separate UI layout from behavior. Just put your code to create and layout widgets in one place, and your code to set signals and slots in another place.
- I hate use the auto-connect SLOTS and SIGNALS feature (based on naming convention such as "on_my_button_clicked".) I have found that I almost invariably have to set up this connection at a determinate time, not whenever Qt does it for me.
- ...

The GUI has been made of several widgets:

3.1 Monitor Widget

The monitor widget is designed to draw the graphical output of the game.

3.1.1 Graphics

The graphical image used in this project are also inspired by the Platinum Rift II game, the original image which was downloaded from CodinGame website is shown in Fig.3.1.

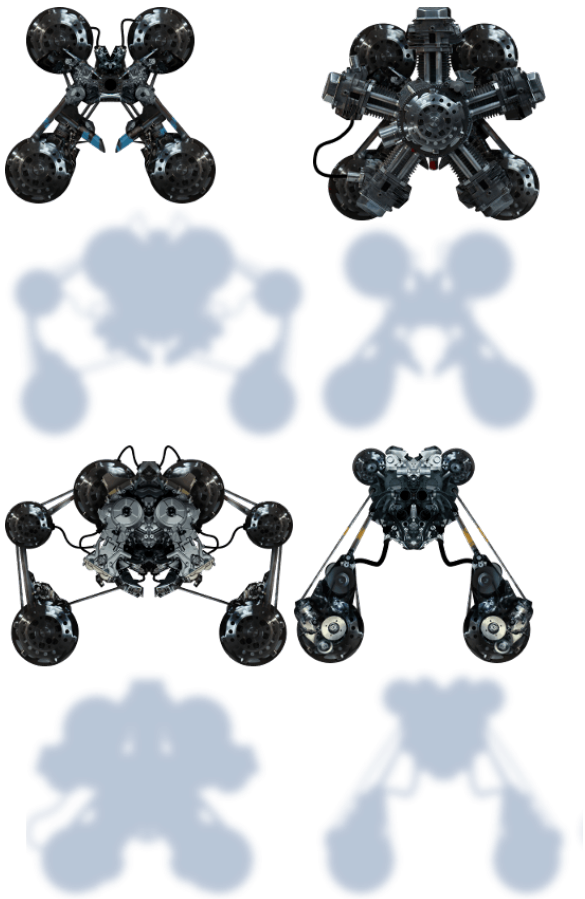

Figure 3.1: Original image

Then using this image and with the help of Adobe Photoshop software[4] the following images were created for different kind of robots and in different colors to use in this project.

3.1.2 Robots

For all four different robots we have in game we needed to create images with 4 different colors, each indicating one team.

Then for drawing the robot in scene we had to draw the shadow first, then the robots engine lights above it, then the robot itself and finally the robot's gun.

EXPLORER

The Explorer robot for 4 different teams with 4 different colors are shown in Fig.3.2.

Figure 3.2: Explorer Robots

The explorer robots engine lights for different teams is shown in Fig.3.3.

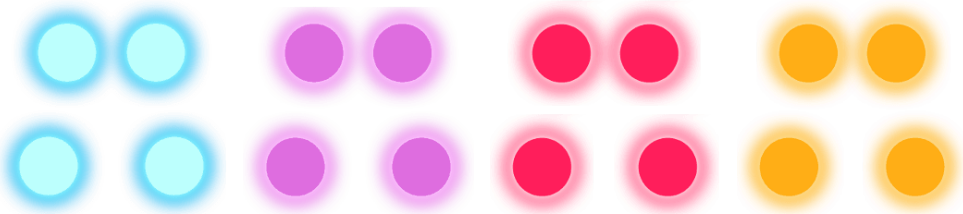

Figure 3.3: Explorer Robot Engine Lights

And finally the shadow of the explorer robot:

Figure 3.4: Explorer Robot's Shadow

TERMINATOR

For Terminator robots as the base robot had same color for all the teams and the difference in color was made in weapons color, we needed one image of robot it self and four images for weapons which are shown in Fig.3.5 and Fig.3.6.

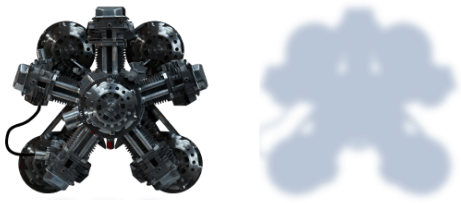

Figure 3.5: Terminator Robot and Shadow

Figure 3.6: Terminator Robot Guns

NINJA

Ninja robots are like the Explorer robots, the difference in color is in their main frame body color.

Figure 3.7: Ninja Robots

And their engine lights are shown in Fig.3.8.

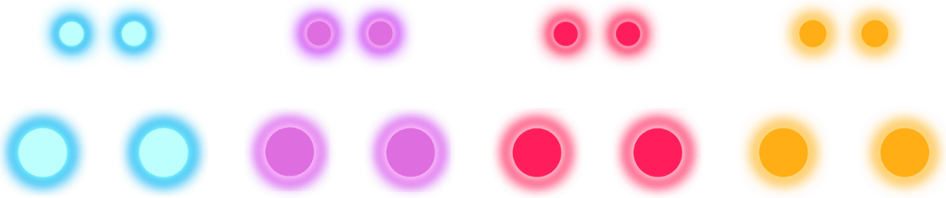

Figure 3.8: Ninja Robot Engine Lights

Finally the most powerful robot in this game is made from these graphics:

Figure 3.9: Predator Robot and Shadow

Figure 3.10: Predator Robot Guns

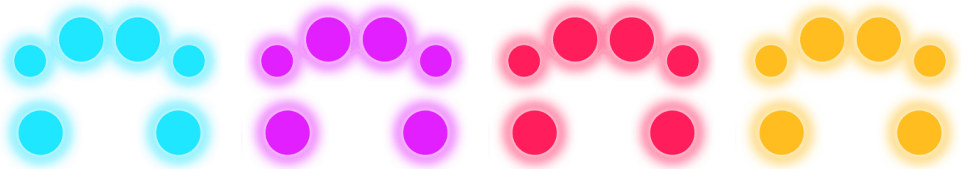

Figure 3.11: Predator Robot Engine Lights

As mentioned before the scene is being drawn in `CMonitorWidget` class, this is done by firstly drawing the map on the scene and then the robots on the map. As we wanted to develop the program just using the C++ and Qt. and not to use any other graphical library such as `OpenGL`, all the drawing is being done on `QWidget` using `QPainter` on the widget. For each robot we had file layers of `QPixmap` drawing above each other. This action was time consuming a lot specially for huge number of robots (600ms for 300 robots). To avoid this problem we draw each robot in each direction and for each color once and saved the 24 resulting images in the corresponding robot's class, thus the problem of frame rate by increasing the number of robots in the scene has been solved.

The Monitor widget is a simple `QWidget` that we use a `QPainter` to draw everything on it. This procedure starts with drawing the background image, then the map above it, and finally the robots on their correct position on the map as shown in Fig.3.12

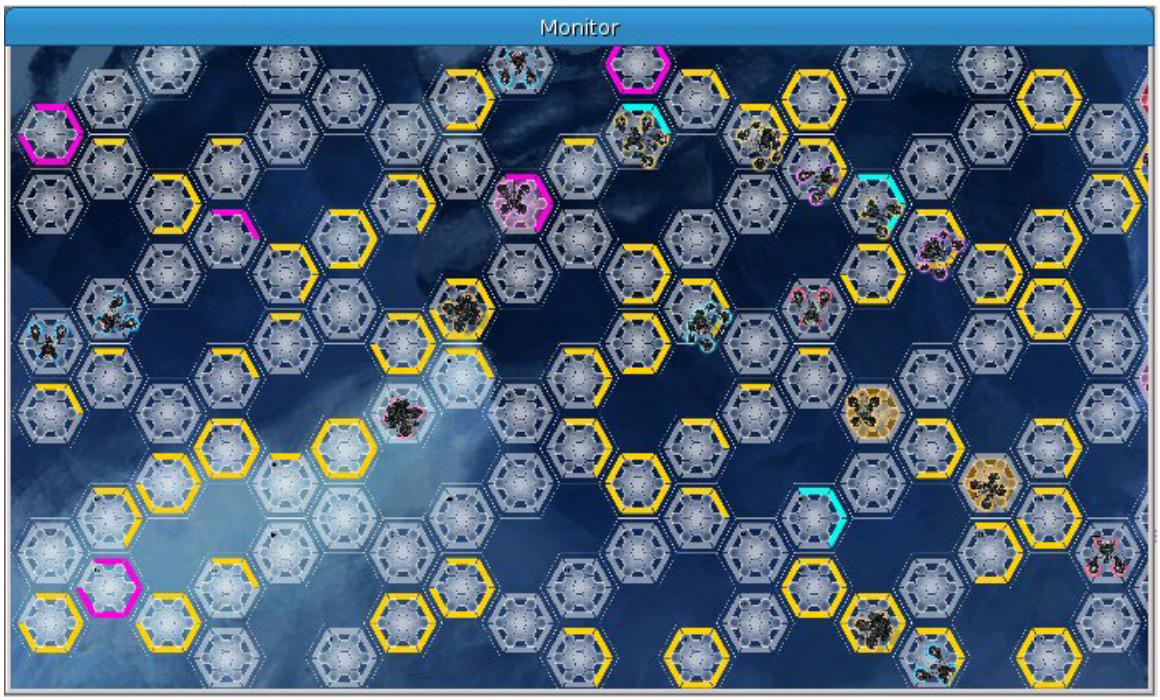

Figure 3.12: Monitor Widget

On this widget we capture the `QWheelEvent` for zoom/unzoom the screen and also capture `QMouseEvent` for drag and drop and navigating in the map.

3.2 Main Window

The main window is a class which's base is `QMainWindow` and creates all the other widgets, it has a central workspace and adds the widget to its workspace to be shown.

CHAPTER 4

DISCUSSION

4.1 FUTURE WORKS

- Animating the Robots movement(will be done by presentation day)
- Create a GUI map generator so that users can generate their own map by some click
- Change the monitor widget to `OpenGL` widget
- Make the Conquer VoidLand cross-platform

REFERENCES

[1] Programming is Fun | Codingame (http://www.codingame.com/).

[2] GCC, the GNU Compiler Collection (https://gcc.gnu.org/projects/cxx1y.html).

[3] Qt. 5.4.x (http://www.qt.io/qt5-4/).

[4] Adobe Photoshop (http://www.adobe.com/products/photoshop.html).

APPENDIX A

GAME RULES

A.1 CHRONICLE

Planet VoidLand - Solar year 963369

Biological life forms have long ceased to exist on Earth. They were wiped clean more than 900,000 years ago by sentient synthetic beings: the Rhoplagoldincons. The Rhoplagoldincons were named by their creator -VoidMinded, the now extinct human- in reference to the Rhodium, Platinum, and Gold, the most valuable chemical compounds critical to the performance of their central nervous system.

The Rhoplagoldincons left Earth on year 2022 at the start of a major period of glaciation. They found a new planet to live on and called it VoidLand and started to populating there. This newly exiled race then divided into four rival factions which now fight each others for survival and universe domination.

Following major tectonic shifts caused by the glaciation, large rifts now abound at the surface of the VoidLand. As the glacial period is now getting to an end, it is time for Rhoplagoldincons to return to Void, to mine and extract Rhodium, Platinum, and Gold and produce more members of their faction.

A.2 THE GAME

A.2.1 YOUR TASK

You are at the head of a faction and your objective is to conquer back planet Void.

You can navigate within the map in the same way you would on google maps: zoom/unzoom with the mouse wheel and move using drag'n drop.

A.2.2 THE PROGRAM

Game is played on a map representing planet Void continents. Each continent is shaped using **hexagonal zones**.

Links between zones are provided at the start of a game as a graph.

When the game starts, all zones are neutral. Rhodium, Platinum, and Gold mines are located on a given number of zones. They produce 0 to 6 bars of Rhodium, Platinum, or Gold per game round.

The Rhodium, Platinum, and Gold are used to buy Robots for fighting.

Players goal is to conquer a maximum of zones using war Robots. Taking ownership of zones allow you to win more Resources (Rhodium, Platinum, and Gold) to buy more Robots. You get ownership of a neutral zone by moving a Robot on it.

You start a game with no Robots and a initial budget, you have one continent placed randomly on the map but never on a zone with Resources (unless the map does not have a continent without resources). You can start by purchasing a Robot with you initial budget and place it on this continent to start the game.

With each game round, the following actions are executed sequentially:

1. First step: **distributing**. Each player receives a number of Resources bars related to the number of Resources available on their owned zones.
2. Second step: **moving**. Each player moves as many troops as they want on the map base on moving rules.
3. Third step: **buying**. Each player buy Bots and place them on the map.
4. Fourth step: **fighting**. Once all players have completed steps 1, 2 and 3, fights are triggered on zones.
5. Fifth step: **owning**. Ownership of zones changes.

A.2.3 RULES FOR DISTRIBUTING:

- Each player receives as many Resources bars as the number of bars available on the zones which belongs to the player.
- These bars add up to the number of bars already mined by the player and not yet converted to Robots.

A.2.4 RULES FOR MOVING:

- A Robot (or group of Robots) can only make one move per game round.
- A Robot can only move from one zone to a contiguous zone âĂŞ either neutral, already owned or owned by an enemy.

- A Robot located on a zone where a fight is ongoing -meaning a zone with enemy Bots on it- can only move to a neutral zone or a zone he/she owns. Simply put, a Robot that flees a fight can only retreat on a zone which does not belong to an enemy.

A.2.5 Rules for buying:

- Each player can buy as many troops as their Resources stock allows.
- A freshly bought Robot can only be placed a zone owned by the buyer.

A.2.6 Rules for fighting:

- A fight is triggered on every zone having Bots from more than one players.
- For each fight zone, damages are being calculated from multi target robots then single target ones. Hit points will be reduced from higher hitpoint robots to lower hitpoint ones.

A.2.7 Rules for owning:

- A zone with only one player's Robots on it is won -or kept- by this player.
- A zone with no Robots on it or with Robots from multiple players on it does not change ownership: it remains neutral or kept by its previous owner.

A.2.8 Robots

There are 4 kind of Robots player can buy with resources:

- **Explorer:** Explorer Robots are cheap their mission is to navigate the map and discover more resources. But they have low hit point so they can destroy easily. Their power is also low, they have **single target** weapon with **low damage point** and can not damage other robots that much, they are powerless in front of other powerful Robots.
  - Name: **Explorer**
  - HitPoints: **60**
  - Weapon type: **Single Target**
  - Damage: **60**
  - Price:
    - Rhodium: **0**
    - Platinum: **0**
    - Gold: **20**

- **Ninja:** Ninja Robots are a bit more expensive than Explorer Robots, they also have more hit points. Although their weapon is also **single target**, but it has a **high damage point** and can destroy other robots easily one by one.
  - Name: **Ninja**
  - HitPoints: **180**
  - Weapon type: **Single Target**
  - Damage: **300**
  - Price:
    * Rhodium: **0**
    * Platinum: **10**
    * Gold: **30**
- **Terminator:** Terminator Robots are even more expensive than Ninjas, this is due to fact that they have **multi target** weapon that can damage all the Robots in a continent and wipe all the explorers in two shots. But their weapon is not that powerful and as mentioned it takes two shots to destroy an explorer.
  - Name: **Terminator**
  - HitPoints: **210**
  - Weapon type: **Multi Target**
  - Damage: **30**
  - Price:
    * Rhodium: **3**
    * Platinum: **20**
    * Gold: **40**
- **Predator:** Predator Robots are the most powerful, most expensive, and as it's obvious from their name the most dangerous robots ever been made. they have a very powerful **multi target** weapon that can destroy all the explorer robots in a continent in one single shot. They also have very high hit points.
  - Name: **Predator**
  - HitPoints: **300**
  - Weapon type: **Multi Target**
  - Damage: **90**
  - Price:
    * Rhodium: **30**
    * Platinum: **60**
    * Gold: **90**

At the end of a game, the player owning the more zones wins.

A.3 GAME PLAY

Your program must first read the initialization data from standard input, then, in an infinite loop, read the contextual data of the game (ownership of zones and location of Robots) and write to standard output the actions for your Bots. A template code is being provided for ease of use in sectionB. The protocol is detailed below.

Your program has 100ms maximum each round to send all instructions to your Robots.

INPUT FOR THE INITIALIZATION PHASE:

Line 1: 4 integers:

- `playerCount`: number of players (up to 4 for now) `myId`: id of your player (0, 1, 2, or 3) `zoneCount`: number of hexagonal zones on the map. Zones are identified with a unique id ranging from 0 to (zoneCount - 1) `linkCount`: number of links between zones - i.e. number of frontiers between zones.

`zoneCount` **following lines:** for each zone, three integers `zoneId`, `resourceType`, and `minePower` providing the number and type of resources bars that can be mined on that zone per game round.

`linkCount` **following lines:** two integers `zone1` `zone2` providing the ids of two connected zones -meaning a movement is possible from `zone1` to `zone2` and vice-versa.

INPUT FOR EACH ROUND: Line 1: three integers `myRhodium`, `myPlatinum`, and `myGold` providing the number of resources you have in stock.

`zoneCount` **following lines:** for each zone, eighteen integers:

- `zId`: id of the zone
- `ownerId`: player id of the zone owner (-1 for a neutral zone)
- `explorerP0`: player 0's Explorers on this zone
- `ninjaP0`: player 0's Ninjas on this zone
- `terminatorP0`: player 0's Terminators on this zone
- `predatorP0`: player 0's Predators on this zone
- `explorerP1`: player 1's Explorers on this zone
- `ninjaP1`: player 1's Ninjas on this zone
- `terminatorP1`: player 1's Terminators on this zone
- `predatorP1`: player 1's Predators on this zone
- `explorerP2`: player 2's Explorers on this zone
- `ninjaP2`: player 2's Ninjas on this zone
- `terminatorP2`: player 2's Terminators on this zone
- `predatorP2`: player 2's Predators on this zone

- `explorerP3`: player 3's Explorers on this zone
- `ninjaP3`: player 3's Ninjas on this zone
- `terminatorP3`: player 3's Terminators on this zone
- `predatorP3`: player 3's Predators on this zone

OUTPUT FOR EACH ROUND:

Line 1: A series of movement commands. A movement command is composed of 4 integers `robotsCount`, `robotsType`, `zoneOrigin`, and `zoneDestination` which indicate the number of robots of any kind to move from one zone to another.

For example `"4 T 2 1 3 E 2 6"` = two commands: moving four Terminator robots from zone 2 to zone 1 and moving three Explorer roots from zone 2 to 6.

Just write `"WAIT"` if you do not wish to make any movements.

One should be careful that all the characters in non numeric inputs **SHOULD BE CAPITAL LETTERS**.

Line 2: A series of purchase commands. A purchase command is composed of 3 integers `robotsType`, `robotsCount`, and `zoneDestination` which indicate the number and type of roots to add to a zone. For example 2 N 32 1 P 11 = two commands: buy two Ninja robots for zone 32 and buy one Predator robot for zone 11.

Just write WAIT if you do not want to buy anything.

Again be careful that all the characters in non numeric inputs **SHOULD BE CAPITAL LETTERS**.

For robots type (model) four letters are used as their initials :

- **'E'**: Explorer robot
- **'N'**: Ninja robot
- **'T'**: Terminator robot
- **'P'**: Predator robot

APPENDIX B

TEMPLATE

A template code is generated in C++ for ease of use by player, it can also be found on GitHub repository by the name `Template.cpp`

```
#include <iostream>
#include <string>
#include <vector>
#include <algorithm>

using namespace std;

/**
 * Auto-generated code below aims at helping you parse
 * the standard input according to the problem statement.
 **/
int main()
{
    int playerCount; // the amount of players (between 2 to 4 )
    int myId; // my player ID (0 or 1)
    int zoneCount; // the amount of zon es on the map
    int linkCount; // the amount of links between all zones
    cin >> playerCount >> myId >> zoneCount >> linkCount; cin.ignore(
    for (int i = 0; i < zoneCount; i++) {
        int zoneId; // this zone's ID (between 0 and zoneCount-1)
        int resourceType; // Neutral=0 Rhodium=1 Platinum=2 Gold=3
        int minePower; // Number of resource zone mine has
        cin >> zoneId >> resourceType >> minePower; cin.ignore();
    }
    for (int i = 0; i < linkCount; i++) {
        int zone1;
        int zone2;
        cin >> zone1 >> zone2; cin.ignore();
    }
```

```
    // game loop
    while (1) {
        int myRhodium; // your available Rhodium
        int myPlatinum; // your available Platinum
        int myGold; // your available Gold
        cin >> myRhodium >> myPlatinum >> myGold; cin.ignore();
        for (int i = 0; i < zoneCount; i++) {
            int zId; // this zone's ID
            int ownerId; // the player who owns this zone (-1 otherwi
            int explorerP0; // player 0's Explorers on this zone
            int ninjaP0; // player 0's Ninjas on this zone
            int terminatorP0; // player 0's Terminators on this zone
            int predatorP0; // player 0's Predators on this zone
            int explorerP1; // player 1's Explorers on this zone
            int ninjaP1; // player 1's Ninjas on this zone
            int terminatorP1; // player 1's Terminators on this zone
            int predatorP1; // player 1's Predators on this zone
            int explorerP2; // player 2's Explorers on this zone
            int ninjaP2; // player 2's Ninjas on this zone
            int terminatorP2; // player 2's Terminators on this zone
            int predatorP2; // player 2's Predators on this zone
            int explorerP3; // player 3's Explorers on this zone
            int ninjaP3; // player 3's Ninjas on this zone
            int terminatorP3; // player 3's Terminators on this zone
            int predatorP3; // player 3's Predators on this zone
            cin >> zId >> ownerId >> explorerP0 >> ninjaP0 >> termina
        }

        // Write an action using cout. DON'T FORGET THE "<< endl"
        // To debug: cerr << "Debug messages..." << endl;

    cout << "WAIT" << endl; // First line for movement commands
        cout << "WAIT" << endl; // Second line for purchase commands
    }
}
```