1 Introduction

In this assignment we will create our own computer game and practice using Modern C++ concepts: dynamic polymorphism, RAII, STL Containers, and STL Algorithms. You will be graded on your software being “simple” (as in not overly complex). Software that is more simple is easier to understand and easier to change. These characteristics are important for software that is in use in the real world as often software needs to be changed regularly. We will use Kate Gregory’s description of simple software from her “Simplicity: not just for beginners” talk, take some time to watch it first.

2 Tasks

Next install and run VirusGame on your computer using the installation instructions provided, have a look at the documentation and code to get somewhat familiar with it. Then work on the following tasks.

2.1 task1: Dynamic Polymorphism

Currently in VirusGame.cpp all Virus units are stored in a static array:

```c++
Virus units[max_nr_units];
```

But we want to be able to add objects of other classes as units besides only the “Virus” objects. In addition we want to handle the “player” object
as just another unit so the code gets simpler. Therefore change the “units” array so that units of different types can be added to it by using dynamic polymorphism.

2.2 task2: Avoid Duplicate Code

Avoid having duplicate code or expressions or said differently: don’t repeat yourself (DRY). The Virus::step() function is currently already a duplicate of Player::step(). Find a good way to avoid that and any other duplication.

2.3 task3: RAII

With dynamic polymorphism you will often have to dynamically allocate memory when you instantiate objects. This could be done using the “new” keyword. However you will then have to remember to deallocate the memory using the “delete” keyword when it is no longer needed to avoid memory leaks. Alternatively with Resource Acquisition Is Initialization (RAII) you can make sure you, and other people that might use your code in the future, won’t forget to release any resource such as memory. This is done by putting the code that releases the resource in a destructor that is automatically called when an object goes out of scope. Use RAII in your code to make sure all resources are released automatically.

2.4 task4: STL Containers

The modern Standard Template Library containers are the preferred data structures to use. Prefer std::vector over a static array as it can grow to arbitrary size, it knows its own size, it doesn’t decay to a pointer when passed to a function, an assignment make a full copy, and has only little additional overhead compared to a static array. Therefore replace any static array in your code (for example: Virus units[max_nr_units]) with a std::vector and prefer STL containers if you choose to add other data structures.

2.5 task5: STL Algorithms

ES.1 of C++ Core Guidelines recommends using the standard library over “handcrafted code”. Therefore use as much as possible the functions
defined in the [STL Algorithms Library](#) instead of for example raw for-loops. For a gentle introduction to STL Algorithms see the [“105 STL Algorithms in Less Than an Hour”](#) talk by Jonathan Boccara.

### 2.6 task6: Your Own Creative Extension

The VirusGame is not yet finished. Extend it so it has interesting game play. Maybe the player has to avoid touching the viruses, or shoot them, or bump into them to bounce them into an anti-virus unit. Maybe also add some special effects like explosions or tire/skid marks or keep a score. The more creative the better, make it fun. You are free to change anything in the provided source code. Write a short description of your extensions with references to your source code and maybe how I should use them during the game in the README.md file just so that I don’t miss anything when grading your submission.

### 2.7 task7 (advanced): SOLID principles

As source code grows from a few hundred lines to many thousands of lines it tends to get harder and harder to change. This is one of the most difficult problems in software engineering. There are different schools of thought on how to alleviate this problem. One of these is using the SOLID principles which focuses on reducing code dependencies. For the SOLID principles see the [“Breaking Dependencies: The SOLID Principles”](#) and [“Free Your Functions!”](#) talks by Klaus Iglberger. Try to use the SOLID principles to decide how to (re)structure your code. Describe the code structure decisions you made in relation to the SOLID principles in the README.md file with references to your source code.

### 3 Grading

Your grade will follow from which tasks you complete to a satisfactory level:
<table>
<thead>
<tr>
<th>task</th>
<th>points</th>
</tr>
</thead>
<tbody>
<tr>
<td>task1: Polymorphism</td>
<td>1</td>
</tr>
<tr>
<td>task2: Avoid Duplicate Code</td>
<td>1</td>
</tr>
<tr>
<td>task3: RAII</td>
<td>1</td>
</tr>
<tr>
<td>task4: STL Containers</td>
<td>1</td>
</tr>
<tr>
<td>task5: STL Algorithms</td>
<td>2</td>
</tr>
<tr>
<td>task6: Creative Extension</td>
<td>3</td>
</tr>
<tr>
<td>task7: SOLID principles</td>
<td>1</td>
</tr>
</tbody>
</table>

Points will be deducted for code that is not simple as described by Kate Gregory.

## 4 Plagiarism

You are not allowed to share code with other students, if we detect (manually or with plagiarism checkers) that different submissions have similar structure we will have to report that to the examination board (see the UvA “Fraude en plagiaat regeling”).

Therefore if you optionally choose to fork the VirusGame git repository, so you can use git to track and backup your changes, then make sure the repository is private otherwise you could get accused of plagiarism if someone copies your code. Bitbucket allows you to make repositories private if you use your “@uva.nl” email address for your profile.

If you base parts of your work on code written by others add references to the source. Code that is not written/designed by you will generally not earn you many points so don’t rely on others too much.

## 5 Submission

Submit your code as a zip/tar of the whole VirusGame project before the deadline on Canvas. Remove the compiled executables and other derivatives that are not needed to compile your code in order to reduce the size. If you add other dependencies (additional libraries) describe them in the README.md so I can easily install them. Double check that your submission contains all required files before you submit.