Virus Containment

A game for Project Multimedia 2008

Bas Boterman, Sander Stolk, Teunis van Wijngaarden
{bboterm, ssstolk, teunis}@cs.vu.nl
Dept. Computer Science, Vrije Universiteit, Amsterdam, The Netherlands

February 22, 2008

1 Introduction

This document describes the usage, configuration, and operation of the game *Virus Containment*. This game – in which the player fights against viruses – is created for *Project Multimedia* at the VU University in Amsterdam. In short, the goals of the developers were to:

- Experience creating a game,
- Learn programming C++, and
- Combining their different developer experiences.

Although the game is not very realistic, it does serve an educational goal. *Virus Containment* is about protecting the (human) body, which is threatened by viruses. Players should send special killers to the viruses in the form of virus containers. Of course, reality is much more complex, the game only touches it.

We – Bas Boterman, Sander Stolk and Teunis van Wijngaarden – are students Computer Science, currently following master courses. Bas and Sander specialize in internet and web technology and Teunis in multimedia. In section 2 we will give an overview of the game, in section 3 we will show how anyone can configure the game. How we programmed and structured the game will be shown in section 4, possible improvements will be discussed in section 5, and finally we will end with our evaluation (section 6).

2 The Game

2.1 Purpose

The purpose of the game is to protect your body by destroying the attacking viruses. Viruses can be made harmless by special immune system defense cells, which we'll call virus containers (or containers for short) from now on. These containers are perfect fits for one specific virus, but not for others: like a key to a lock, or in this case the other way around, like a fitting lock around a key. When a virus comes into contact with a container, the virus is pulled towards the container and is then made harmless, not being able to reproduce anymore. If a virus manages to get near actual human cells (the tissue), it has the ability to infect it. The virus penetrates into such a cell and reproduces until the host cell explodes. The viruses are harmful to a human's body and need to be neutralized. Therefore we would like the user to use the containers as the viruses' enemy. In figure 1 you can see which container can neutralize which virus. All pictures have a very distinctive shape and color for easy matching. Plus each container looks exactly like a perfect fit for its matching virus, which is of course the purpose they were designed for.

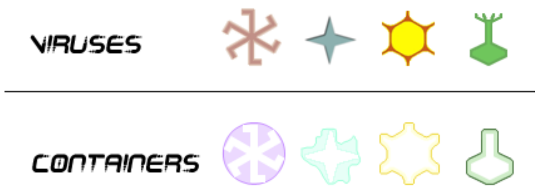

Figure 1: Present viruses and their containers.

2.2 How to play

The main menu offers three options: starting a new game, viewing the highscores, and viewing the credits. Since the last two options speak for themselves, we will only discuss the game below.

Every level starts with a playfield filled with a set of viruses (Figure 2). The amount of viruses and their positions are pre-defined for each level. The bar at the top of the screen provides some useful information:

- **Sent out** tells you how many containers are active in the playfield and how many are allowed in total.
- **Time** is the time remaining to fight the viruses. This differs for every level.
- **Score** is the score, which increases after destroying a virus.

Furthermore, there are icons of containers on the bar. Click one of these icons to get the indicated container in the playfield. To catch a virus, drag the container in the virus' path. When the virus is close enough, the container pulls the virus towards it. The proximity differs for every type of container. Catching a virus results in a higher score! When you destroyed all viruses, you'll procede to the next level.

When a virus hits the tissue, there's a chance on infection. That is indicated by a dark spot on the tissue. After a while – depending on the type of virus – the tissue opens up and more of the same viruses enter the playfield. Don't let this happen, because it takes you a lot more work to fight all those viruses. When your time is up, or the amount of viruses on the playfield is too large to fight, you are game-over. If you did a good job, you are asked to enter your name for the highscore. Check the highscore to see if you can call yourself one of the best *Virus Containment* players.

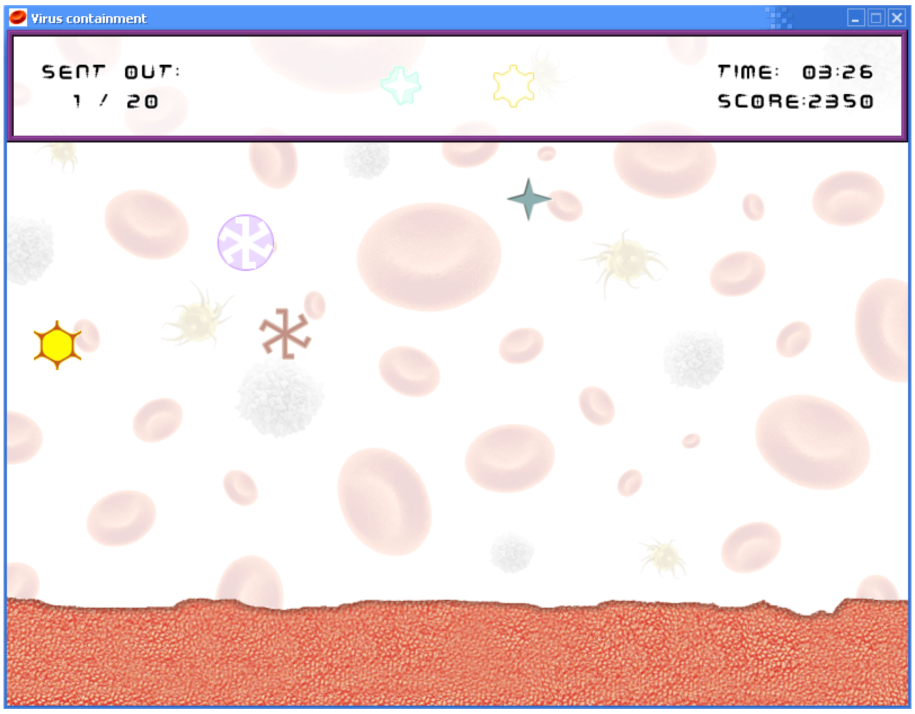

Figure 2: A playfield with a set of viruses.

3 Configuration

The game can be adjusted without having to change the code and rebuild the program. This makes the game structured and it gives players the opportunity to easily create new levels or adjust the virus'/container's behavior. Hence, no programming skills are needed.

```
[TIME]
Limit = 120

[CONTAINERS]
Limit = 20

[VIRUSES]
Number = 3

[VIRUS 1]
Type = 4
Positionx = 100
Positiony = 100

[VIRUS 2]
Type = 3
Positionx = 300
Positiony = 100

[VIRUS 3]
Type = 2
Positionx = 400
Positiony = 250
```

Figure 3: A level setup.

3.1 Creating Levels

The configuration of the different levels can be found in the directory: `/assets/data/`. Levels are named `Level #.ini` (where `#` is the level number). The game reads in levels in numerical order until there are no more level files in row (after which it assumes the game is won).

The ini-files contain information about the time, the maximum number of containers, and the type and position of viruses. The example file (Figure 3) shows a level with:

- A time limit of 120 seconds, thus 2 minutes.
- A maximum container sent-out limit of 20.
- Three viruses: one of type 4, one of type 3, and one of type 2.

The Number of viruses should always be consistent with the configuration of viruses below. If not, the behavior of the game is undefined. Also, always stick to the format in the files of the original game (also the order!). Try to find a balance between:

```
[GRAPHICS]
Normal = assets/graphics/Virus 1.bmp

[AI]
PathLength = 1/8
ToTissue = false

[MOVEMENT]
Speed = 50

[INFECTION]
Rate = 0.3
Multiplier = 10

[SCORE]
Containment = 50
```

Figure 4: A Virustype setup.

- Time limit and number of viruses, i.e. the more viruses, the more time is needed to contain them.
- Container limit and number of viruses, i.e. the more viruses, the higher the limit.
- Number of viruses and their types, i.e. do not put many nasty viruses in one field.
- Type of viruses and container limit, i.e. when you put viruses with high multiplication in the field, also choose a high container limit.
- Type of viruses and y-position, i.e. position a virus that heads directly to the tissue, high in the playfield.

3.2 Change Virus Behavior

The definitions of the virus type configuration can be found in the same directory as where the levels are stored. Currently, there are four virus types that describe four different looks of viruses and their type of behavior. Figure 4 shows an example virus type:

- The `Normal` variable in the `[GRAPHICS]` section tells the game what picture to use for the virus. The path can be relative to the game.
- The variables beneath `[AI]` define the direction to move. `PathLength` tells the game how often the virus changes direction. `ToTissue` defines if the virus heads directly downwards to the tissue or not.

```
[GRAPHICS]
Normal = assets/graphics/Container 1.bmp
Icon = assets/graphics/Container 1 Icon.bmp

[MOVEMENT]
Speed = 10

[CONTAINMENT]
Distance = 15
```

Figure 5: A Containertype setup.

- In the `[MOVEMENT]` section the `Speed` is defined in pixels per second.
- The `Rate` variable in the `[INFECTION]` section gives the change that the virus will infect once it hits the tissue. If it does not infect, it will bounce. The `Multiplier` variable in this section defines how many viruses will be created after an infected cell explodes. The time for the cell to explode dependeds on this variable too.
- The `ContainmentScore` in the `[SCORE]` section defines the amount that the player's score is raised, when he or she contains this virus.

Again, please stick to the format and order in this file. Try to find a balance between:

- `PathLength` and `Speed`, i.e. let viruses with a short path length move fast.
- `ToTissue` and `Speed`, i.e. don't let viruses that head directly to the tissue move too fast.
- `Infection Rate` and `Multiplier`, i.e. if the change of infection is low, choose a high multiplier ("it will almost never infect, but IF it does...").
- `AI`, `Movement`, `Infection`, and `Score`, i.e. give a high score for nasty viruses.

3.3 Changing Container Behavior

Although containers do not move on their own, like viruses do, they need some configuration (Figure 5). As explained above containers can only catch viruses that have the same type number.

- Besides the `Normal` graphic that containers have in their `[GRAPHICS]` section, there is also the `Icon` variable. This is the graphic that will be displayed on the bar.
- Containers also have a `Speed`, defined in the `[MOVEMENT]` section.
- The `Distance` variable in the `[CONTAINMENT]` section defines how near (in pixels) a virus should be, in order to be destroyed by this container.

Again, please stick to the format and order in this file. Try to find a balance between:

- The nastiness of the virus and the containment distance, i.e. make a nasty virus a bit easier to catch.
- The speed of the virus and container.

3.4 Changing Graphics, Music, and Sound Effects

All graphics and audio are stored in subfolders of `/assets/`. To change the graphics, bitmap files (`*.bmp`) from the `/assets/graphics/` folder can be replaced by another bitmap file and these will be loaded in the game without recompiling. We have chosen to use the RGB color `#FF00FF` as transparent background color (as can be seen from figure 6). Same goes for music and sound effects, these are in the folders `/assets/music/` and `/assets/soundeffects/` respectively.

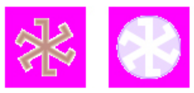

Figure 6: A virus and a container with transparent background.

4 Behind the Game

Virus Containment is programmed in a structured way, using classes. Every class is placed in a separate file, so please check the source directory to get more insight. In the header files is defined what a class can do. This section describes the main structure of the game, and discusses some essential elements in more depth.

4.1 Global structure

Below the global structure is described by explaining the most important classes:

Main This class is executed first. It sets some variables that have to do with graphics, and it calls the `draw()`, `logic()` and `input()` functions of `Game`.

Graphics This class handles all the graphics. Every class that has a `draw()` function makes use of the `Graphics` object.

Game This class keeps track of the game state, which is splash screen, main menu, win screen, lose screen, or ingame. The `logic()` function of `Game` just calls `logic()` of the class that is used for the current game state. The same goes for `draw()`. `input()` routes the input to the class of the current game state.

MenuMain When the main menu is the current game state, `logic()` and `draw()` of `MenuMain` are called by `Game`. This class handles everything that has to do with the menu. The other menus do a similar thing.

Ingame Ingame is used when a new game is started. It loads a level and sets variables in `Info` and `Playfield`. After that, it calls the `logic()`, `draw()`, and `input()` function of `Playfield` and `Bar`.

Info The object of this class is the administration-object. It contains information about the current level, the score, etc.

Playfield This class contains a list of viruses and containers that are active in the game. It calls the `logic()` and `draw()` function of every entity that is in the game. They act individually. Ingame can ask `Playfield` if the game is lost or won. `Playfield` knows this by checking the number of viruses and time remaining.

Bar This class is dedicated to display information (time, score, sent out, etc.) in the top bar.

Tissue This class makes the tissue appear at the bottom of the playfield. It contains an array of infected cells and can answer the question if a virus collides with the tissue.

Virus A Virus moves all on its own, because the position is changed randomly by `logic()`.

Container A container's position is determined by the users input. That input is routed to `Playfield`'s `input()` function, which alters the path of the container dragged. The containers ask the playfield for a list of viruses. When a virus gets near the container, he takes over control. He pulls the virus to his own position and kills it.

EntityType, ContainerType and VirusType The characteristics of viruses and containers are determined in the `.ini` files. These files are read in during the game, and the information is stored in one of the classes above. Because `VirusType` and `ContainerType` share some properties, they have the same superclass `EntityType`.

Sound This class contains some basic functions for playing/stopping music and sound effects.

Other Finally there are some classes that do not need explanation because they speak for themselves, or do not play an important role. Those classes are: `Position`, `TimeStamp`, `Splash`, and `Font`. Hence, we won't discuss these classes any further.

4.2 In Depth

4.2.1 Menus

The `Menu` class is responsible for all menus in the game. There are various menus to be seen. These are the main menu, a win menu (i.e. you won the game) a lose menu (i.e. game over) and a highscore menu (i.e. you played so well you may enter your name for the hall of fame).

We have one class, which is the `Menu` class. This is an object which all other menus use. It contains functions for drawing the nescessary graphics and code to process input events like keys that are pressed on the keyboard and mouse clicks, which are generated by the user. Also it provides a comfortable means to place text on the various menus.

The main menu (which can be found in the class `MenuMain`) is responsible for the first menu after the splash screen. It's not a very spectacular menu, since it only displays the three options as stated above. The same goes for the win and lose menu.

The highscore menu is a little bit different, since it has to load various high scores from a file (`assets/data/Highscores.sav`) and displays them on screen. Also it has the means to save new highscores by using the `saveHighscores()` function.

4.2.2 Ingame, Playfield and Viruses

`Ingame` is where the actual game happens. One (static) instance of `Ingame` is used, and the function `getIngame()` can be used to access that instance. Don't forget to include `ingame.h`. The same construction is implemented for `Game`.

With the initialization of `Ingame` (by the constructor) the `Info` object is initialized and the containertypes and virustypes are created. The latter use `.ini` files as their input.

After initialization by the constructor, the `init()` function of `Ingame` is called. This function (re)sets the bar and playfield, and calls `readLevel()` to read a level from a level `.ini` file. This function sets variables in `Info` and adds viruses to the playfield.

Now, the bar and playfield are set up and the game is ready to start. From now on the sequence is: draw, input, logic. This sequence is started by `Main` and, if the gamestate is `INGAME`, passed to `Ingame`. What do these functions do in `Ingame`?

Draw Just draws the bar and playfield.

Input Routes events to the bar and playfield.

Logic Calls the `logic()` function of the bar and playfield. This function also checks if the game is won or lost and acts upon that by changing the gamestate.

In fact, besides setting up the game and keeping the reference to `Info`, the `Ingame` object just routes events and calls `logic()` and `draw()`. This structured view makes programming easier. Let's discuss the bar's functionality.

Draw Gives the bar's information to the graphics object. Therefore it reads the information (score, time, sentout, etc.) from the `Info` object, adds a position, and passes it through to the graphics object.

Input The player must be able to click the container icons. This function handles the click events on the bar and puts new containers on the playfield.

Logic The Bar has no more functionality, so the `logic()` function is empty.

The `Playfield` is the other class that plays an important role. This `Playfield` holds the tissue and a number of containers and viruses. Besides the initialization functions, the following functions are interesting.

Draw Viruses and containers act on their own, so `Playfield` calls the `draw()` function for every container and virus. It also calls `Tissue`'s `draw()` function.

Input This function handles the dragging of containers. When the mouse button is down, it adds positions as a direction.

Logic Again, just like `draw()`, this function calls the logic function of all the containers and viruses. In addition the `Tissue logic()` function is called.

Containers and viruses have a lot in common, that is why they both have `Entity` as their superclass. The classes `Container` and `Virus` almost only have a `draw()` and `logic()` function. Let's first discuss some variables of their superclass.

Containers and viruses should move time based. This prevents them from moving faster on faster PCs. In order to accomplish the time based movement, their old position and the "time they were there" is saved in a variable (in their superclass `Entity`). The current position can now be computed based on the current time, the position, the time where it started, the position where it moves towards to, and the speed. It is done like this:

- Compute the distance with the first position and the destination position.
- Compute the total amount of time required to travel that distance, using the speed of the entity.
- Compute the time currently travelled divided by the total amount of time required to travel to the destination.
- Use that calculated fraction, to determine the entity's current position, which is between the first position and the destination position.

Because a virus or container cannot destruct itself, we store if it is alive or not; this is done with the boolean variable `alive`. The playfield then checks this variable once in a while and deletes the entity if needed.

The last important variable in `Entity` is the path. This vector contains positions to where the entity should move. For the container these positions are determined by the mouse movement (dragging). The virus sets the positions in the path randomly. Classes outside the entity can add positions by using the `addDirection()` and `setDirection()` functions.

The `draw()` function in both `Container` and `Virus` does nothing more than just writing the image to the output on the current position. The logic functions are more interesting:

Entity's logic This function changes the positions based on the speed, time and path.

Container's logic It calls the entity's `logic()`. Besides that it checks the virus vector in the playfield on nearby viruses. If there indeed is a virus in the `containDistance`, it makes the contained variable point to that virus and it deletes the virus from the playfield. If the container already contains a virus, it does not check for near viruses. It changes the direction of the virus in order to get it at the same position as the container. Finally the container adds some points to the score, kills the virus and kills itself.

Virus' logic Again the entity's `logic()` function is called in order to change the position. It checks if the virus hits the tissue and if it infects or bounces. After that the bouncing (on border or tissue) is handled. The way to do this is by mirroring the path. Finally, a random direction is added to the path if needed.

There are more classes used. Please check the source code for more information about them.

4.2.3 Simple DirectMedia Layer (SDL)

The Simple DirectMedia Layer (SDL)[1] provides access to input and output devices, such as the mouse and keyboard or screen and speakers. Using SDL has two clear advantages, due to it providing a layer of abstraction over a system.

- It means there is no fiddling with too low-level functionality which saves some production time.
- It makes it easier to port an application or game when an implementation of the SDL library exists for the target system as well. If that is not the case, it might be possible to only replace the SDL functions with system-specific function calls and successfully recompile that way.

SDL also has a couple of disadvantages though.

- The graphics structure of SDL is not that user friendly, but is thankfully not really needed unless one wants to do more with graphics than just displaying it – such as editing the graphics and checking color values of pixels.

[1]The SDL project website: `http://www.libsdl.org/`.

- It provides no layers for graphics, meaning programmers either have to implement these themselves by using separate drawing buffers and later combining those, or by drawing all the objects in the order of back to front from the get-go (which might not always be easy and/or possible). It would be a good improvement if it were to support layered graphics itself in a clear way.
- Without the aid of extra libraries, only loading in BMP and WAV files are possible. This is not that bad, actually, if it weren't for the fact that SDL does not mix loaded in soundfiles together into a single playable stream. Effectively this means that one can only play one sound at a time unless you aren't afraid to try mixing soundfiles together. Luckily there are other libraries to complement SDL, such as SDL_mixer which can provide the mixing of audio. However, this not being present in SDL itself, and is disturbingly lacking.

The installation process of SDL and its libraries is somewhat awkward and required some fiddling; mostly because we haven't had much experience with linking in dynamic libraries. After it being set up though, it was not that difficult to figure out its interface and use thanks to the nicely documented SDL API which can be found on its website.

The use of SDL can be seen in main.cpp for setting up the window and main drawing surface, in `graphics.cpp` for another abstraction of loading in BMPs and drawing them using SDL and in `timestamp.cpp` for requesting the amount of milliseconds passed since the start-up of the program.

4.2.4 SDL_mixer

Since mixing sounds in SDL is not very easy to do yourself (i.e., you have to combine wave files yourself, but there's no means to know where to start mixing since you don't know where the audio buffer currently is with playing), we chose to use the SDL_mixer library, written by Sam Lantinga, Stephane Peter, and Ryan Gordon.[2]

SDL_mixer provides some handy functions when it comes to mixing audio samples with, for instance, music files. After including the `SDL_mixer.h` file we can use functions to initialize the audio device (`Mix_OpenAudio()`) and play music and sounds (`Mix_PlayMusic()` and `Mix_PlayChannel()` respectively). Stopping music is done with `Mix_HaltMusic()`, and finally we can clean up the memory `Mix_FreeMusic()` and close the audio device (`Mix_CloseAudio()`).

Although this library is very convenient, the documentation is quite brief. It took us quite some time to install it properly (we had to figure out to use both the precompiled libraries and the header files, from two different

[2]The SDL_mixer project website: `http://www.libsdl.org/projects/SDL_mixer/`.

archives from the website). After this worked we could build nice functions like `playMusic()` and `playSound()`.

5 Future Improvements

Although the final version of *Virus Containment* works well and is fun, it is always possible to improve. These improvements will probably never be realized, because the university project has finished, but it is worth mentioning some of them.

First of all there are some basic features which we left out of the game. These include sound settings, pausing the game, and other miscellaneous settings/adjustments a user could do in the game.

The dragging of containers could be made smoother. It would also help the game player if the path that the container plans to travel is visible. Also the random path of the virus could be made smoother.

The levels, viruses and containers (their types), could be balanced better. This would cost hours of game playing, because there are a lot of possible configurations. We as developers also discussed about decreasing score when infection happens. Or changing the balance between some variables (see section 3).

Although the 'oldschool' look and feel of the game is attracting to a certain (gaming) public, improvements in graphics and sound could make it a better game. This would make the (educational) topic more realistic and attract a broader public.

A nice feature that has not been implemented because of the time, is interaction between viruses and containers. It would be nice if viruses react on containers. For instance, really 'smart' viruses could move away from their container. Then the player has to force the virus in a corner of the field. Another improvement could be that containers 'see' viruses and move towards them. This 'auto-pilot' function could help in lower levels.

A great enhancement would be porting the game to another platform or device. This might add some fun; imagine playing *Virus Containment* on the Nintendo Wii or DS!

Making *Virus Containment* three dimensional would be really cool. This would probably require reprogramming of almost the full game (graphics class, all the graphics, positions etc.).

6 Evaluation

We all have a different background and therefore different experiences in programming. Teunis had programmed C once before, but only for a university course. Bas already had more experience with C and also a bit with basic C++. Sander knows C++ also, and uses C a lot in his free time. We all had in common that we never programmed advanced C++, and never created a game in this fashion.

From the start the tasks were divided in a natural way (Table 1). Sander knew most about C/C++ and had a clear structure in his head. Bas and Teunis had more basic problems with programming and left the overall structure over to Sander. We wrote the structure down with the three of us. This made it a lot easier to divide tasks and split up.

Bas:
Program helper classes
Program fonts
Create graphics and sounds
Program sound
Sander:
Invent game structure
Program graphics
Program menus
Improve ingame physics
Teunis:
Program ingame (virus, and container)
Adjust ini-files
Write documentation

Table 1: The tasks. Obviously we cooperated, but the one responsible is listed.

Sander did the really nasty part: the first steps in SDL. Bas followed him and did the programming of sound and music. He also created some helper classes (i.e. TimeStamp, Sound, Font, and Info), did the graphics and sounds. Teunis focused on programming a part of ingame (and the classes it uses). He also adjusted the `.ini` files, created levels and wrote great part of the documentation. This project took us approximately 2 months of parttime work.

6.1 Personal Evaluation

6.1.1 Bas

Having above average programming skills in C and basic skills in C++, I was quite blank about what to expect. I had never programmed beyond the console, so using SDL was new to me as well. A great challenge! Also, I am a fan of old fashioned C, which is in great contrast with object oriented languages like C++: it's a complete new way of programming.

I had to get used at using objects again. All the things like public/private classes, constructors, and inheritance were very new to me in C++. This was a great learning experience for me (although we still had to use pointers and other workarounds, because C++ lacks some things – see below). Other than that, I've learned a great deal of looking at Sander's way of structured programming and organizing the entire game.

Finally it was the first time I used third party libraries like SDL. Although Sander already implemented great part of SDL, I had the honors of doing so with SDL_mixer. I spent a well afternoon figuring out what to download, compile, link, and execute. Finally, after some clever trying, I succeeded. I'm really proud of what we have achieved in the last two months and it was fun working on other elements of the game besides coding as well (graphics and sounds).

6.1.2 Sander

Creating this game proved to be more time consuming than expected. Then again, pretty much all (programming) projects are. Me having a small bit of experience with developing games, made the structure of development not part of the learning process for me. Things I learned the most out of this project were using SDL, using timestamps to base logic on instead of expecting logic to be called at the exact same intervals, and the use of the programming language C++.

I was quite pleased with SDL itself. Not having to use platform-specific calls for creating a window and drawing to the screen saves a lot of time and effort better spent elsewhere. C++ started out as a nice and refreshing programming language for game development, but in the end, due to it being somewhere in between C and Java when it comes to features, I do not like it as much as I was hoping for. Programming using objects in C++ is still a bit awkward, and the lack of a garbage collector makes it harder to use, especially when using references instead of pointers as that makes it more difficult to see where variables actually exist.

6.1.3 Teunis

I had some experience programming C, but I forgot everything as time passed by. I even forgot I used C once. So I had to learn C++ almost from scratch. I've read a few chapters in a C++ book about the structure of C++ programs, data types, classes, arrays and pointers. Those where the language specific topics that were new to me. Other things were not that hard, because I've already programmed more advanced Java.

Most difficult to understand was the notion of pointers. Using objects and arrays was not easy for me and I must admit that in the beginning I sometimes used the asterisk (*) on a trial-and-error base. Another difficulty is understanding the nasty compiler messages. The unclear messages made it hard to detect the cause of an error.

Overall I found learning C++ was not very hard, because of my knowledge of other programming languages. Though I must say that programming more advanced things (like with SDL) would still be too difficult for me. I find C++ more difficult than Java, because in C++ you have to handle more 'low-level' things. I think that Java is more suited to learn object oriented programming, because it is more about the logic (not about the language). On the other hand, I experience Java as slower. Also, a great advantage of C++ is that it runs (on Windows) without installing an extra 'layer'. This results in my conclusion that C++ is better in practice and Java is better in theory.

I think in the group work worked out very well. It would take me days to program the graphics part, but Sander did it a lot faster. This made it possible for me to program the logic of viruses and containers without having to deal with the difficulties of SDL. Programming the logic and things like that where easier for me, because I understand the programming logic. The thing I had to learn here now was the language C++ and the game structure. Afterwards, Sander changed some things in order to make the code a bit better.

Writing a great part of the documentation and cleaning the code was useful for me. It made me look at the final code and re-evaluate it. Although I'm not a game player, I liked creating the game. It is an interesting type of application. It is nice to learn from the structure/logic of a game.