Rendonan Minigame

Software Engineering Large Practical

Word Count: 4600

Teun Kokke, s1242775

December 8, 2014

General Concept

My project is the combination of a website, database and a turn-based game, with the game itself at its core. Users are able to register an account on the website, requiring only a username and password, and can then immediately jump in to play the game.
Players start with a basic set of stats: *experience, money, health, strength and agility.* From this point the player is expected to fight the monsters that will continuously re-spawn upon death. By killing the monsters, the player earns both money and experience, where the money can be used to upgrade stat points, making the player stronger and able to combat harder, more rewarding monsters. The experience plays a major role in the user ranking and contributes to leveling up the user, improving his looks.
When the player leaves the game, the game progress will be saved and re-loaded when he/she returns. However, when the player dies, all progress will be lost. Only at this point, the stats will be stored in the highscores.
The highscores can be viewed on the website, and contain the stats of each death that occurred for players. The ranks are decided stat-wise, each stat can be selected and filtered individually in ascending/descending order. Additionally, each player can be sought for manually, in the search-bar.

> *Due to my excitement for software engineering, and in hope to improve my mark, I successfully managed to find an abundance of time and made it well-spent.*

files

- Report: report.pdf
- Documentation: documentation.txt
- Game Sourcecode: Open files with any text-editor, optionally with syntax highlighting. The language is GML but Javascript highlighting will suffice. Files: /GM_SOURCE/sourcecode/rendonan.gmx/...
  - Objects: /objects
  - Scripts: /scripts (structure displayed in Figure 3, "Difficulties" section of this report)
- Website:
  - routing: /src/Rendonan/MiniBundle/Resources/config/routing.yml
  - controllers: /src/Rendonan/MiniBundle/Controller/...
  - entity classes (db tables): /src/Rendonan/MiniBundle/Entity/...
  - views: /src/Rendonan/MiniBundle/Resources/views/...
- Tests:
  - test coverage: /testcoverage/index.html and testcoverage/dashboard.html
  - test results: /testresult

Structure

Website

In order to explain the code structure, it is useful to be able to relate to the expected result and then match this to the code requirements. For this reason, the following section looks similar to a "user guide".

Design

Coming up with the following design was pretty straight-forward. In order to play a game, as a specific user, and be able to see the highscores, on a website, the website is divided into 4 (user-interact-able) pages: Main, Game, Highscore and Registration. Each of the pages are assigned with a specific controller, and each controller contains actions for specifying the state the page is in.

1. Starting from the ***main page*** (MainController), a few things can be noticed:
   (a) There is a menubar on top of the website which is present at any of the pages in the website.
      - On the left the text displaying "Rendonan" is a link sending the user back to the main page.
      - On the right, either a login form is displayed when the user is not logged in, or the username of the logged-in user along with a link to logout or delete his/her account.

   (b) There are 3 links in the main content section: Play, Highscores, Register.

   (c) There is a simplified highscore table (loaded from the database) displayed on the right.

2. From the main-page, by pressing the url to ***register*** an account the user will be forwarded to the registration page (RegisterController). Here, if the user was already logged in , he/she will be notified of this. Otherwise 2 forms will display, one to register, one to login. Both forms only accept alphanumeric strings between length 4 and 20 characters. Upon successful registration, the user is immediately logged in with the given username and redirected back to the main page.

3. From the main-page, the ***highscores*** (HighscoreController) link can be clicked. These scores are primarily ordered by score and then by experience (in descending order).
   But it is possible to filter the results based on purely experience / wealth / health / strength / agility in both ascending and descending order by pressing the '^' or 'v' buttons.
   Additionally, specific usernames can be sought for using the user-search-bar right above the highscore table.

4. From the main-page, the ***play game*** (GameController) link can be clicked to open the game page.

   (a) If the user was not logged in, he/she will be immediately redirected to the register page, implying that he/she must first register and then try again.

   (b) If the user was logged in, the game will automatically start loading and running.

   - When the game starts, the user's data (user that is logged in) will be queried from the database and then received by the game. The user will be held in a waiting room until this data was succesfully received by the game. Otherwise the game client will repeatedly attempt to establish a connection.
   - Once the game has successfully started up and loaded the user data, the game will start and allow the user to play.
   - The user's progress will be automatically saved to the database. This event occurs every 5 seconds. Therefore when the player stops playing the game, he/she can proceed where he/she left when returning to the game in the future.
   - When the player dies, the progress is sent to the highscores, and will then be reset for the user such that the player has to restart the game completely.
     All that remains from his past game is the score, which (depending on how good was relative to the other scores) will be visible in the highscore page.

   Additionally, a ***404-page-not-found*** error page (PageNotFoundController) exists in order to notify the user that a requested page does not exist.

Having a manually generated error-404 page happens to be an important factor for the website page ranking score for well-known search engines.

Symfony follows a systematic approach to rendering html responses, which in simplified form looks as follows:

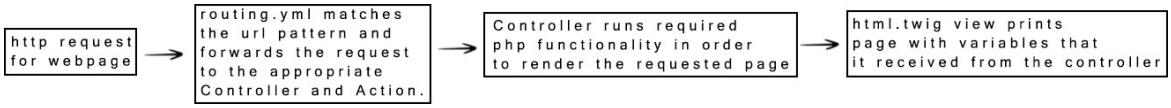

Figure 1: Displays the back-end path from a http request to generating the appropriate response.

Following this structure, generating a new page was done by first defining the url in the routing, which connected it to the right controller and action. The controllers take care for generating the required output, with each of the requirements being mentioned in the points above. Aside of the pages that are visibly rendered for the user, some actions contain pure logic blocks, e.g. user logout and game-load/save. These will execute the appropriate logic commands (typically session and/or database related) and then redirect to a different controller action that actually does render a user-friendly view.

Refactoring

As for refactoring the website, little refactoring was required. This is because I did not have to write many re-usable "modules". The main task here was to set-up the routing in order to direct the right url to the right controller with the right action (function). The code used inside the controllers was generally quite unique for each page, as each page is indeed unique on its own.
The only common factor is the user-session, for which I created a separate class. This class contains a method *get_sessionData()* which returns an array containing all the useful user data. If at any point I wanted to store a new piece of information about the user, only this class would have to be modified, and then every page would immediately have access to it.
Another common factor would be to retrieve the highscores from the database (required in highscore and main-page, potential for more).

However due to a chain of complications with the framework I did not manage to successfully create a separate module for this. (This matter is further discussed in the deficiencies)

Game

Design

Similar to the website, the game is expected to log a user in, allow the user to play, stop the game when the user dies, and be tested by the developer when required. Therefore it is separated in 4 individual rooms / views: Connect, Game, Death and Test.

1. In the ***connection room***, the controller object is instantiated, which controls the entire gameflow. At this point, its main tasks are to check whether the game should run in game-mode, runtime-test mode or unit-test mode.
   (a) If the unit-test mode was set, the controller will immediately forward the game to the test-room.
   (b) Otherwise, the controller will attempt to connect to the database in order to fetch the user stats. Once the user stats were received (using *http_get()*), it will forward the game to the game-room.
2. In the ***game room***,
   (a) The controller:
   - creates stat-upgrade purchase-buttons for each of the upgradable stats
   - creates the player instance
   - respawns the monsters: *monster_respawn()*
   - draws the user interface: *draw_UI()* (stats menu)
   - draws the combat menu: *draw_menu()*
   - tracks the user stats
   - controls game event timers
   - forwards the game to room_Death when the player dies (hp below 0), and submits the stats to the savescore highscore controller.

(b) The stat-upgrade purchase buttons (obj_Button) use the player stats (stored in the controller) to decide the cost of the upgrade, the amount by which the stat is upgraded and does the arithmetic that comes to play when pressed.

(c) The player instance (obj_Player) draws the player as entity with appropriate image based on the level which it calculates using *calc_level()* after synchronizing with the stats from the controller using *player_update()*

(d) The monster instances (obj_Monster)

- Draw themselves as entity with appropriate image based on level (set randomly by a user-defined range in the combat menu upon creation by the controller).
- Calculate their own combat stats based on their level
- Create a message upon death, incrementing the controller's xp and money stats (*monster_death()*).

(e) the user interface (drawn by obj_Controller) displays the stats of the player stored in the controller object.

(f) The combat menu (drawn by obj_Controller)

- Displays a charging attack-bar based on the controllers attack-cooldown timers (influenced by the agility stat), which when filled allows the player to attack.
- Draws an *attack()* button when not in auto-combat mode, which can be pressed in order to attack the monster.
- Draws an auto-attack button which toggles the auto-combat mode variable in the controller. When auto-combat mode is on, the attack-bar charges slower, but the attack button will disappear as the player will automatically *attack()* the monster when the bar is filled instead.
- Draws a potion button which can be pressed to heal the player slightly at the cost of money.
- Draws a *monster_controller()* which includes in-/decrement buttons for the future monster's re-spawn-level range.
- Draws a special-skill (*draw_sunflash()*) charge bar which when filled displays an activation button, which in turn triggers a particle system object that damages the monster.

Due to the complexity, here is a visual representation showing the game room:

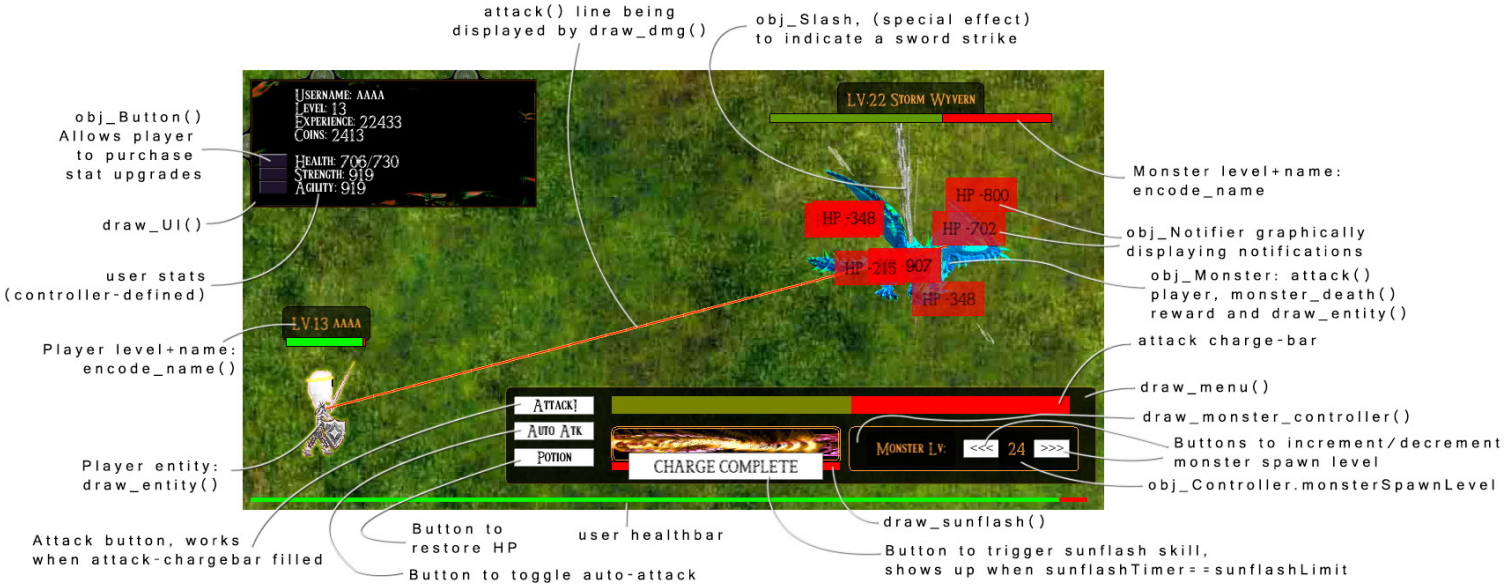

Figure 2: Visual representation of the game room, displaying many of the modules mentioned above.

3. In the ***Death room***, the controller draws a simple message notifying the user of his/her death and waits for a user input-response to restart the game.

4. In the ***Test room***, the controller instantiates a test-object *obj_TestConnection*, which:

   (a) Executes all unit tests for the functions used throughout the game (*test_exec()*)

   (b) Tests if it can connect to the server (php controllers)

   (c) Tests if it is successful in its attempt to load user data

Refactoring

Little "RE-factoring" was required as I have spent a lot of time working on projects in Gamemaker in the past, having given me the experience and

knowledge of many potential threats and issues that may be lurking in the near (and even distant) future. Therefore many of these issues were (mostly unconsciously out of habit) treated before they became an issue.
It should indeed have become apparent that the controller object sets the entire game in motion. Every instance is responsible for only itself, and is at most dependent on the controller, of which only exactly 1 instance exists at any time, meaning that this dependency is harmless.
Due to this design, it is unlikely that modifying any module breaks any other module (as long as I (or whichever programmer) sticks to the rules of the module in question, meaning that for example the return type should not change).

Function Structure

Since Gamemaker uses an internal xml-like structure in the main game-project file in order to organise the functions into sub-folders; but on the harddrive only stores the scriptnames in one single folder, here is an image the structure:

Difficulties

Before the development started (around the time of writing the proposals) I (successfully) predicted to get stuck for a relatively long period of time on exactly 3 points:

- Since we were tasked to behave like real "Software Engineers" as opposed to simple "programmers", I took it on me to learn using an industrially accepted framework, "Symfony", for the website. In the past I succeeded to teach myself how to program websites (including back-end, registration / login system etc.), starting blank and developing the entire set-up and structure. It was complicated using the framework as I was not "allowed" to use the majority of the methods I was used to using in the past. For this reason, it is possible that I have (unknowingly) written my own functions for tasks that Symfony could have solved instead. However for obvious reasons, I had to find a

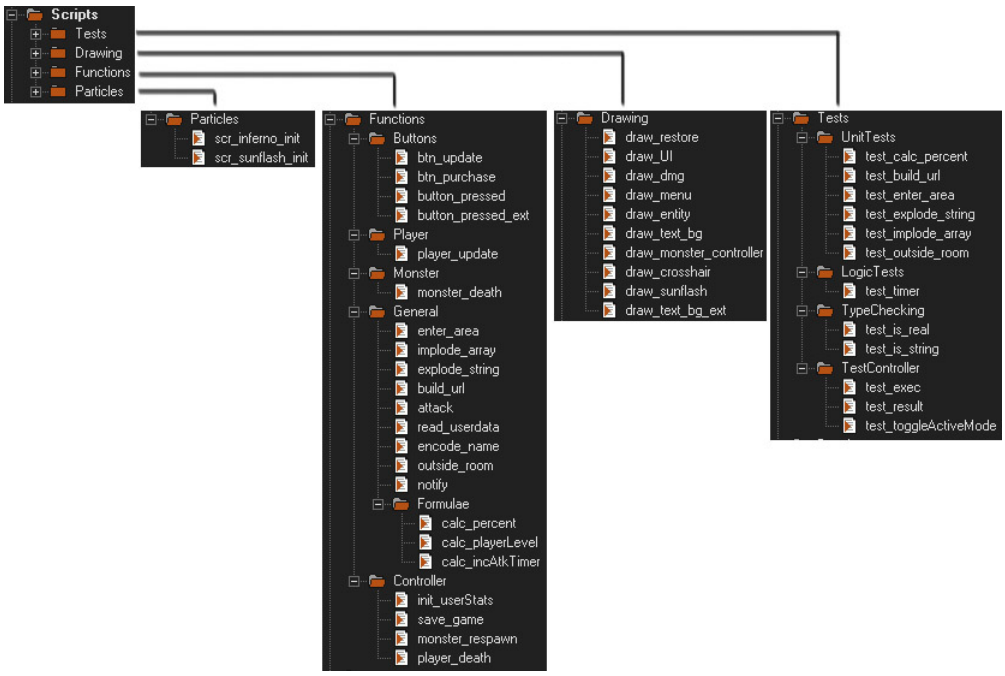

Figure 3: Script Hierachy, displaying the alignment of scripts used in gamemaker. Their content is located in the folder GM_SOURCE/sourcecode/rendonan_release.gmx/scripts.

proper balance between "spending development time" and "spending research time".
The research was time-spent with no visible progress, yet, having awareness about getting stuck at many points during the development due to the lack of knowledge was something I did not look forward to.

- The user log-in system of Symfony is completely different to what I am used to working with. From my past experience, logging in a user was as easy as setting session-variables (id, username, whichever). Symfony however, appears to use some sort of firewall security system which tracks each user. I have spent days in attempt to understand how this works, tried many different suggested setups but all failed. The clock kept ticking so in the end I gave up trying to cooperate with the framework on this matter.
  Therefore, plan B, I used "sessions" directly (or rather, a Symfony-

accepted version of them). This method worked, although it did mean that users are not truly recognised as "users" by the framework. Although I designed my own system for this in a way that enabled me to still have the required control over the users.

- Connecting GM to the Database using http_get is something I have been stuck on for a long time in the past.

  It was one of the main reasons why I have never made browser games, but rather downloadable clients which allow the use of dll files and c++ socketing (allowing even multiplayer functionality).
  It took me several days to solve this issue, having created a separate project just for the sake of experimenting with http requests. It required me to run a software update for Gamemaker, after having purchased an additional module for the software which allowed me to include external JS scripts. With such JS script, I triggered an asynchronous Ajax call to the website (url defined in the routing.yml, which then loaded the desired (php) controller), which would query the database and return the results back to the game.

Testing

Website

My apologies for waiting so long for researching on how to do testing in the php framework. This was partially due to my lack of imagination on what could be tested, as well as the fact that I had only little understanding of how the framework itself works to begin with. This added sufficient complication on its own, to not immediately demand for further research on how this could then also be tested. However if I happened to have contributed to the "bus-factor" before my tests were written, this would have been a bad excuse for making a high-end company lose revenue.
On a positive note, I did eventually succeed in installing a testing unit called "phpunit", with for I wrote tests for the following:

- each of the pages were tested to load correctly
- When no page was specified by the user, he/she will automatically be redirected to the main page

- When a non-existing page was specified by the user, a manually generated 404-page-not-found message should be displayed.
- The registration and login form validations were tested for "bad" input (e.g. quotes, semi-colons, unaccepted string-length, etc.).
- The error-messages returned by the form validation
- Successful registration tested to indeed be successful
- Account deletion tested to indeed delete the account
- Adding a user highscore was tested using a long sequence of tests, starting from registering the user, logging it in, saving new game data, letting it "die" to store the data into the highscores, ensuring the game score was reset, (removing the highscore entry again), and removing the user.
- The load and save-game scripts were tested which the game will be using to access in order to load and store data onto the server.

Game

Tests were also written for parts of the game engine (front-end). Since Gamemaker itself does not come with any testing modules, i created my own, and have split them up into 4 sections (visible in Figure 3):

1. ***Type Checks:*** Since I used many functions, each function call has been embedded with type-checks to ensure that the input is of the correct type, making sure that when calling a function it has been given proper arguments. Since GML is a dynamically typed language, not knowing a function has been given a wrong argument type may cause bugs that are difficult to spot.
2. ***Logic tests:*** From personal experience, logic errors are the most commonly occurring, most difficult to spot and therefore also most time-consuming to fix. The best method of testing if the game logics work correctly is generally to actually run the game and see if everything happens the way it is expected to.
   However, I did create some methods that display in-game variables

(that would otherwise be hidden). Being able to see the actual values of these variables is one of the fastest ways to spot if something is wrong in the game logics.
Most potential for these errors lies in event triggers and erroneous timers. So these are displayed when the test-mode is active.

3. ***Unit tests:*** Each function that takes input and has a return-value can be tested with unit tests. Therefore each function that matches these criteria is considered with unit tests.

4. ***Connection tests:*** A more complex type of test, but also one of the more important one. Since the game is expected to connect to the database, tests are written to confirm that the game indeed manages to successfully connect to the server, and manages to receive user data.

I have created my own testing module to execute the latter 2 testing types. This will trigger when the test_mode in the controller object is set to unit-testing. At this point, all of the unit tests will be executed, as well as the connection to the server.
If at point in time, any of the tests mentioned above fails when either in test-mode or unit-test mode, a warning message will be displayed.
The controller object also contains an "active-testing" variable (which can be toggled by pressing spacebar), while in testing-mode. When active, even tests that passed will return a message, confirming that the test was executed and passed.
If the game is in regular game-mode, of course none of these messages will pop-up. The game would execute as if nothing happened.

Test Coverage

Of course everything in the project (modules, database, connections, user interfaces etc.) could be tested to a much greater extent, both the website and the game. However for the level of this project, the amount of testing done roughly matches the requirement.
One of the more important things that could be tested further is testing the game-save script from the game's point of view (as opposed to what is currently only done from the website's perspective). This would require the POST-request sent by the game, to also expect to receive a return value about whether or not the request was received successfully. The same concept

applies for storing the user's score into the highscores table.
Another (very interesting) test would be a complete system test, not only covering either the website or game separately, but both at the same time. Registering a user-account, making it log in, start-up the game, let the game execute in an automated-mode, expecting it to save the score (check if this works), expect the bot to die at some point and see if the score was saved, then going to the highscore page to confirm the score is there, and then delete the score and the user account.
This system would cover the typical lifetime of a standard user.
Meaning that if this test passes, it can be expected that users following the same pattern won't have any issues either. The only problem is that writing this test would involve the server sending an additional value to the game stating that it should run in automated mode. Although most of this is indeed already done as the player can enter auto-attack mode, the point of converting from web-platform testing to game-testing (and back) adds complication as intuitively neither of the elements actually know of eachothers existence.

Deficiencies

I do not believe I have made any large "errors" in my decisions when planning out the setup of the project elements. All in all I feel like I have followed a rather smooth and consistent workflow without large drawbacks, of which, partially I am glad as it means that my system works. On the other hand I am slightly worried that this may also imply that I have unknowingly fixed issues, meaning that I would have failed to inform developers which don't have the experience that I have about threats I discovered and fixes that I made.
The biggest issues I have stumbled upon are, again, the lack of knowledge on the Symfony platform from my part. Although it is arguable whether this is due to a "bad decision" or an expected side-effect of a choice to learn industrially-accepted software engineering standards for PHP development.
Either way, the deficiencies I am most aware of are:

- Having repeatedly failed to use Symfony's methods of generalising database queries, referred to as "entity repositories". I had made many different attempts, used forums, read the manual etc. in order to figure out how these repositories were supposed to be used. I managed to

use them for standard queries, but in order to give it arguments I had to re-write the entire query with just that argument being different. Therefore after a long struggle for several times I gave up and decided to run the queries directly inside the controllers that rendered the pages (queries for fetching highscore data and user data).

- As mentioned previously, the user-login system for the Symfony inbuilt-system was another major drawback as I made many attempts but still could not get this to work. Also here I was forced to fall back to my old knowledge of using sessions.
- In Gamemaker, it is not possible to define a function inside a script. Instead, it expects the user to create a new individual function script, and the name of the file will be the name of the function. Arguments passed will be referred to as "argumentx" (with x being any number from 0 up to 19). For an engineer that is used to programming in languages such as Java and C++ this may seem weird at first.

Also, during the beginning of the project I managed to write identical commit messages (eg. "Janitorial work" without explaining details to in what context this janitorial work was executed). However after having found out that this is bad practice I have immediately recovered from these mistakes. Sadly however, it is not allowed to modify previous commits (other than amending last commit made).
It may be obvious, by my commit history, that I have spent more time than the required 75 hours. This is however not due to "bad planning" as I was constantly planning ahead on the way. For each item that I wanted to add to the project, I compared how much time I had left, and how much time I expected to still need for implementing future items. Based on this I was able to make a judgement on the level of detail at which I could work out the item in question, ensuring that I would spend as much time as I could, without actually running "behind schedule". If at any point I would have happened to run out of time, I would still be able to shift priorities to "the absolute necessary" in order to make the project, at least at a basic level, work.

Future Improvements

Choice of technology / language

The reason why I like php is due to its simplicity. It does not require any compiling and works on almost any webserver. However I learned that using a framework such as Symfony removed the simplicity, and it is no longer supported on "any" webserver either.
Therefore using php does not have any more value when planning to build relatively large and stable web platforms than other languages that I know.
Writing plain JavaScript, especially complete games, can become messy very easily. However using Gamemaker solves this issue perfectly.

Future project potentials

There are many ways in which this project can improve, and at high speed. I have attempted to follow an as solid implementation as possible, enabling the ability to build new elements on top of this foundation very quickly and easily.
Purchasable items can be added to the game, along with an inventory system. Monsters may get special skills, requiring the player to pay more close attention to the game during play. Adding chain-attacks that test the user's personal game- and reaction skills, making the game not sorely based on in-game "stat points" in order to be a "good player". Graphics could be greatly enhanced, along with new special sound and visual effects and shaders, contributing to the game's "wow-factor", etc.
(Note that as these improvements are being mentioned, imagining their actual implementation would not be very complicated. They are merely "bricks" that can easily be put on top of the existing project without having to re-write previously defined features.) The website could become much more interactive, by adding visual and technical improvements. Aside of having plain highscores, it could be possible to compare user stats.
Perhaps it is possible to make the game multiplayer over the network entirely, allowing players to cooperate fighting monsters, although this would require further research in order to find out how to solve delay and server/client-firewall issues (especially when using a UDP connection), to which every browser may respond differently.

Conclusion

It is safe to say that by the time I got accepted to Edinburgh University, I had already spent well over 1000 hours programming in Gamemaker. This comes both with up and down-sides.

No-one has taught me how to work with it, I did all the experimenting on my own and have thus far come to the conclusion that the structure of building-up projects in the order I have used for this project is the most efficient and requires least post-re-factoring. By "this structure" I mean to say: creating a controller object and let it take care of everything. Let it store all the important variables of the user, create instances on demand, and allow these instances to depend on the controller if required. But not make them dependent of anything else.
Like writing a book, every new feature (be it a menu, chatbox, attack event, mathematical formula to calculate a stat) should be written in one piece (individual sections, possibly with subsections, eg. draw_menu() with subcomponents), on one location as separate function, preferably being completely independent of everything (other than the input arguments it received).
This way, it is possible to "zoom in" on each part (/component), and be able to edit it without having to worry about anything else breaking down.
Having this experience, it allows me to very easily and quickly create new projects without having to think of it too much. Also, as I manage to stay consistent in my system, I don't have to change my mind as often, reducing the amount of refactoring done.
On the other hand, I developed the system based on my own experiences. Other people may have had other experiences and therefore have come up with a completely different system. I may become blind to certain issues as I don't consider them "issues" anymore: I have become used to them and have my own ways of dealing with them, whereas other programmers may not find it as obvious. This is an issue that heavily relates to software engineering, which is solved by having a well-written documentation.