Software Engineering Large Practical

Teun Kokke, s1242775

December 6, 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.

> *In attempt to explain the code modules behind the project, a "User Guide" section is included to this report for both the website and game. This guide displays the result of the code, and can thus be referred back to in the module section, making it more clear what the code is there for, and what it really does.*

Structure

Website

Design

Coming up with this 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.

1. Starting from the ***main page***, 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 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. 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*** 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*** 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.

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.

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, 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

(b) The stat-upgrade purchase buttons 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 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

- 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 displays the stats of the player stored in the controller object.

(f) The combat menu

- 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 (*attack()*).
- 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.

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 to no "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 are lurking in the near (and even distant) future. Therefore many of these issues were (mostly unconsciously) 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).

Difficulties

Before the development started (around the time of writing the proposals) I successfully predicted to get stuck for a relatively long 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 (unconsciously) written functions for tasks that Symfony could have solved instead. However for obvious reasons, I had to find a 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 is 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 point.
  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 download-able 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

test strategy: how tested implementation. tested enough?

Deficiencies

Deficiencies justified and caused by errors in decision making

issues with using Symfony modules. eg entity query repository, user-login via firewall and security shit

Future Improvements

Things I would do differently: choose the right language/technology? voor herhaling vatbaar?

Mention this would be an option given the time: create a bot to play the game, such that the entire system can be tested throughout: register, login, play game, bot playing game, game repeatedly updates bot stats, bot dies, see if score resets and highscore is saved. check highscore etc.

In the testing room, the test-object could also test whether saving data to the server is successful. This requires a response from the server, which has to then be picked up again by the game.

Conclusion

conclusion: am I happy with what I have? 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 one is the most efficient

and requires least 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 the controller 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, 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 (/module), 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, meaning that when refactoring becomes truly necessary most often the best solution is to actually rewrite an entire function. Luckily this rarely happens.
On the other hand, I developed the system based on my own experiences. Other people may have had other experiences and therefore have 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.