CS5003 — MASTERS PROGRAMMING PROJECTS

CS5003 PRACTICAL 2: GEM HEIST

Deadline: Friday the 3rd April 2020 at 21:00

Credits: 33% of the coursework (and the module)

MMS is the definitive source for deadline and credit details

You are expected to have read and understood all the information in this specification and any accompanying documents at least a week before the deadline. You must contact the lecturer regarding any queries well in advance of the deadline.

AIMS

The main aim of this project is to write a complete web application including both client and server components. More specifically, it will involve designing and testing an API, implementing the front-end and back-end in JavaScript which communicate through this API, and choosing the right data representation.

Your client-side JavaScript should be written in modern JavaScript (e.g. ECMAScript version 6+) with appropriate HTML, CSS and any libraries you find useful. Your server-side should be written in Node.js, using specified packages such as Express. You will be provided with a package.lock file specifying which packages are permitted. Do **not** install and use any other packages without asking the lecturer first. Do not use any other language, JavaScript dialect or templating language (e.g. TypeScript, CoffeeScript, HAML, Pug, React, Vue, ejs, etc.).

The use of git version control system is mandatory for this practical. You will be expected to use it throughout the development to keep track of progress.

OVERVIEW

You will create an online game that allows different users to connect and to play against each other. You will implement *Gem Heist*, in which players take it in turn to disable *alarms* that protect *gems*. Each gem is protected by four alarms. Each alarm can protect up to four gems. The player who disables the last alarm protecting a gem collects the gem and gets to disable another alarm. The game is over once all the gems have been collected. The winner is the player who collects the most gems. Note: *Gem Heist* is a custom game based around the classic Dots-and-Boxes game https://en.wikipedia.org/wiki/Dots_and_Boxes.

Working in pairs, you will implement both the client side (HTML+CSS+JS) and the server side (JavaScript based on Node.js), The server side will implement a RESTful API for exchanging data with the client. Your webpage will contain client-side JavaScript which makes HTTP calls to the API, and exchanges data with the server using JSON.

The client should be able to let a user connect to the server, disable an alarm, and display the current state of game, including whose turn it is and the number of gems each player has collected. Information about which

alarm has been targeted should be sent to the server via the API, and the game state will be held centrally on the server. Your solution should allow two people to play against each other from two separate computers.

The API should provide services needed to make this happen including, but not limited to:

- starting a new game
- configuring the game (player name, shape/size of the grid, number/type of gems)
- disabling alarms
- automatically collecting gems when all their alarms have been disabled
- detecting victory conditions

You should ensure your application provides the core functionality as described the Requirements Section.

REQUIREMENTS

Your application should provide the functionality described below. Many of these requirements are open to interpretation and there are many different ways to fulfil them. You are encouraged to speak the lecturer if you have any doubts or questions about any of the requirements. You should make sure you have completed all the requirements in each section before moving onto the next.

BASIC

Your application should provide the following:

- Allow a player to start/join a game. Note: you may wish to assign each game and each player in the game a universally unique id. The idea is similar to the user ids you used in the StacsYak assignment, but the ids could be hidden from the user and only valid for the length of the current game.
- Allow a player to see the current state of their game, including the gems and alarms in the grid, and each player's current score.
- Allow a player to disable an alarm.
    - If the alarm was the last alarm protecting any gem(s), automatically 'collect' the gem(s) (i.e. remove them from the grid), update the player's score, and allow the player to disable another alarm.
    - If this was not the last alarm protecting any gem, allow the next player to disable an alarm.
- Finish the game appropriately when all gems have been 'collected'.

INTERMEDIATE

Your application should provide all the requirements described in the Basic Section, plus the following:

- Allow a player to enter their name. Note: you can assume the name is lost when the user refreshes the page if you wish.
- Display in the grid, who has collected each gem.
- Keep track of a player's wins and losses and display the current user's statistics to them.
- Allow a player to choose the size of the game (i.e. the number of gems per row/column).
    - Detect if a game is a draw when a game ends.
    - Keep track of a player's draws, as well as their wins and losses.

- Allow a player to leave a game without refreshing the page. The player who leaves loses and the remaining player wins.

ADVANCED

Your application should provide all the requirements described in the Basic and Intermediate Sections, plus the following:

- Allow a player to chose the number of players in a game. Make sure you consider the implications on win/lose/draw scenarios, and players leaving games early.
- Allow non-rectangular grids, e.g. triangles, hexagons, randomly placed gems, etc. Make sure you consider how to let the user specify the grid type, size, and number of gems on the grid.
- Introduce different types of gems, worth different amounts when collected. Make sure you consider if/how to let the user specify what different gem types are used and in what proportions.
- Detect if a player has been inactive for a set period of time during their turn (e.g at least a minute). If they have, assume the player has left the game, and make the remaining player the winner regardless of the score.
- Allow games to be recorded and replayed.
- Allow players to register/login/logout. Any new games they play should be added to their all-time score record. You can assume all records are lost when the server is reset.

DELIVERABLES

A single `.zip` file must be submitted electronically via MMS by the deadline. It should contain:

- The entire working copy of your repository, containing the source code and revision history.
- A **joint** report (around 2000 words), in **PDF** format detailing the design of your solution, discussing any requirements and design decisions taken, your approach to testing, your approach to team work (e.g. use of tools such as Trello, or techniques such as pair programming), detailing the allocation of work within the team, and reflecting on the success of your application and development process. Try to focus on the reasons for your decisions rather than just providing a description of what you did. **This will be joint work of both partners**.
- An **individual** report (around 1000 words), in **PDF** format describing your own contribution and your own challenges.
- A short README file describing how to run your server.

Submissions in any other formats may be rejected.

MARKING CRITERIA

Marking will follow the guidelines given in the school student handbook:

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/feedback. html#Mark_Descriptor

Your submission will not be evaluated based on aesthetic appeal (this is not a visual design course), but use of CSS and DOM scripting which enhances the experience and interactivity will be rewarded.

The mark will be based on the quality of the final application and joint report. This mark may be adjusted for each member of the group based on their individual contribution to the project. Individual contributions will be evaluated using the individual reports and git repository, and will take into account the amount and complexity of features tackled by the individual and how well they worked in the team.

Some specific descriptors for the **group component** of the assignment are given below:

- A **poor implementation in the 0 – 7 grade band** will be missing nearly all required functionality. It may contain code attempting a significant part of a solution, but with little success, together with a report describing the problems and the attempts made at a solution.
- A **reasonable implementation in the 8 – 10 grade band** should provide some of the functionality described in the Basic Section, and demonstrate reasonable use of HTML, Javascript and Node.js. The code should be documented well enough to allow the marker to understand the logic. The report should describe what was done but might lack detail or clarity.
- A **competent implementation in the 11 – 13 grade band** should provide all the functionality described in the Basic Section, demonstrate competent use of HTML and CSS (e.g. for layout and positioning), allowing players to play a complete game against an online opponent. The code should be documented well enough to allow the marker to understand the logic and should have a modular design. The report should describe clearly what was done, with good style.
- A **good implementation in the 14 – 16 range** should provide all the functionality described in the Basic Section and some or all of the functionality from the Intermediate Section. It should demonstrate good code quality, good comments, modular design, and proper error handling. All of the JSON objects passed through the API must be checked and validated. Good use of CSS for layout and styling is expected for a submission in this range (not unmodified defaults). The report should describe clearly what was done with some justification for decision, with good style, showing a good level of understanding.
- An **excellent implementation in the 17 and higher range** should provide all the functionality described in the Basic and Intermediate Sections, and some or all of the functionality from the Advanced Section. It should demonstrate high-quality code and be accompanied by a clear and well-written report showing real insight into the subject matter.

Note: For this practical you do not need to invent your own extensions. Concentrate on providing high quality, sophisticated implementations of the requirements in this specification and writing an insightful report demonstrating understanding.

It is anticipated that in most pairs, individuals will receive the same mark. However, where there is a substantial difference in contributions marks may be adjusted up or down. The following are examples of when the mark may be adjusted.

- A **poor contribution** will have
    - a missing or weak individual report, demonstrating confusion and misunderstanding of the topic. Little or no contribution to the final software, adding little to no value, focussing on only one part of the program functionality.
    - OR a sizeable contribution to the code but a failure to engage in the teamwork process.
- An **excellent contribution** will have an excellent individual report, regular and sizeable code contributions throughout the project, contribution to many important parts of the project, and clear engagement in the team working process.

WORD LIMIT

An advisory word limit of approximately 2000 words for the group report and 1000 words for the individual report, excluding references and appendices, applies to this assignment. No automatic penalties will be applied based on report length but your mark may still be affected if the report is short and lacking in detail or long and lacking focus or clarity of expression.

A word count must be provided at the start of the reports.

LATENESS PENALTY

The standard penalty for late submission applies (Scheme B: 1 mark per 8 hour period, or part thereof):

https://info.cs.st-andrews.ac.uk/student-handbook/learning-teaching/assessment.html#lateness-penalties

GOOD ACADEMIC PRACTICE

The University policy on Good Academic Practice applies:

https://www.st-andrews.ac.uk/students/rules/academicpractice/

HINTS AND SUGGESTIONS

Start by spending some time on design – what needs to be done to meet the basic specification. What type of data needs to be represented and where (e.g. client or server). What is the best way to represent the data – objects, classes, JSON objects, etc. Write a basic outline of your object/class structure and RESTful API on paper and think whether it makes sense. It pays to spend time with your partner and discuss this in detail before starting to code.

Divide the work into small chunks and decide how to split the work between the two of you. Agree on a rough work plan, and meet regularly to see how the work is progressing – sometimes you will need to adjust because things turn out to be more difficult than planned.

Have the basic requirements working fully before venturing into intermediate or advanced requirements. A clean, fully functional and well-written basic implementation is better than a buggy mess with extensions.

Avoid splitting the work into parts that are mostly separate (like only client-side and only server-side) and working on them independently. Integrating such work at the very end is likely to be difficult. Try to work on related aspects – like splitting the API between the two of you, or working on different aspects of the user interface. Then integrate work at regular intervals so you can look at each other's code from time to time. It will help you understand your partner's thinking, help spot problems, and help you pick up good habits and tricks from each other.

Make use of existing resources where available – but make sure that there are no licensing issues and that you credit external sources. A very useful resource for graphics is https://openclipart.org/.

FINALLY

Don't forget to enjoy yourselves and use the opportunity to experiment and learn! If you have any questions or problems please let me know (ruth.letham@st-andrews.ac.uk) — don't suffer in silence!

Author: Dr Ruth Letham

Module: CS5003 – Practical 2: Gem Heist