Project Descriptions

This document describes the deliverables for your projects, and also a list of projects to choose from.

Project Deliverables

Regardless of which project you go for, you must have the following:

1. All work must be in a GitHub repository for the project.
    a. It should be hosted on one group member's account. It should be private, and your instructor should be given read access.
    b. The other member(s) should be added as collaborator(s).
2. You must use the issue system provided by GitHub.
    a. Almost all commits must be linked to issues via the "Ref #. . . " or "close #. . . " keywords.
    b. Almost all issues must be assigned one or more labels.
    c. You must use milestones to organize the issues into sub-goals.
    d. Issue title and description must sufficiently and succinctly discuss the pertinent problem and the overall plan for addressing it.
3. Modularize your code.
    a. Your project should be logically broken into smaller modules that are as independent of each other as possible.
    b. Each module should be in its own file.
    c. Modules should be individually testable as much as possible, and mocks/-fakes should be used when needed.
    d. Functions/methods within modules should as much as possible have a single purpose.
    e. There should be an index.html file that starts your application. Opening that page in a browser should get everything working.
4. You must create suitable commits.
    a. Each commit should address a single issue. To the extent possible, refactorings should be done as separate commits.
    b. Small issues should be resolved by no more than 2-3 commits. You may make separate commits with the tests for the issue and with the implementation/solution.
    c. The code should be passing the linter before you make a commit.
5. Use a linter.
    a. Set up a .eslintrc.json file in your home directory.
    b. You may use the same rules discussed in class, or choose other sets of rules.
    c. Your code files must pass the linter.

d. You can decide if you want the test files to also pass the linter.

6. Write tests.
   a. Unit tests should be arranged in a `test` directory with filenames ending in `.spec.js`.
   b. Tests should be broken into multiple files, organized in some logical fashion (e.g. one test file per code module).
   c. Tests should be written **before** the code they address is written.
   d. Tests should be loaded via a `tests.html` main page.
   e. You may have other files that are used for UI testing.
7. Write documentation.
   a. Your project should have a `README.md` file in Markdown format, describing the project and acting as a user's guide.
   b. A separate `architecture.md` file should describe the high level architecture of your application. What are the different files/modules, what are their relationships? A graphic is encouraged.
   c. Your project should include a `package.json` file including at the very least a name, short description, list of collaborators and a link to the repository.
8. Pair programming.
   a. You are expected to follow a pair-programming[1] paradigm when working on this project. This entails two people working on the same screen, one (driver) writes the code while the other (navigator) observes and considers the relation of the code to the rest of the application.
   b. You should switch roles frequently (no more than an hour or so of coding without switching).
   c. You should always commit and push when you are done, and pull before you start.
   d. Each user should be logging in and committing from their account. All collaborators should end up having a roughly equal amount of commits.

Projects

The following is the list of projects to choose from. You should discuss with each other and rank them. We will randomly generate a group order and then each group will pick a project.

1. SimpleRPG
2. PlaySET
3. LinkedListVis
4. SortVis
5. Shapes
6. Memory Game
7. 2048

[1] https://en.wikipedia.org/wiki/Pair_programming

SimpleRPG

In this game you can create “characters”, and form parties of up to three characters to do combat. It is an extremely simplified role-playing game.

Description

1. A character has the following attributes:

   Name ~ The character’s name plays no particular role other than to distinguish them from other characters. Characters are stored in memory using their name. Once created, the character’s name cannot change.

   Experience ~ You gain experience by winning battles. The amount of experience you gain depends on the power of your opponents, divided by the number of people in your party.

   Level ~ Every character starts at level 1. As you gain experience you advance levels. You need experience points equal to 50 times your current level in order to advance to the next level.

   ```
   Each time you level, you gain a number of "level points", that you can allocate towards
   ```

   ```
   Your chance to hit your opponent is determined by the difference between your level and
   ```

   Health ~ How many “hit points” you have. You have a maximum health based on your level. You lose hit points when you are hit in combat. You return to your maximum health when combat ends. If you reach 0 health you are out of the combat.

   ```
   Everyone starts with some amount of max health, and there are some rules about how many
   ```

   Strength ~ Determines how much damage you do to your opponent if your attack hits. Every point of strength takes away 0.5 of the opponent’s health.

   ```
   Everyone starts with some amount of strength, and strength grows based on "level points
   ```

   Race ~ You can choose one of three races. *Dwarves* start off with a more health and gain more health per level point, *Elves* have a naturally higher chance to hit by their attacks (say 10%) and always go before non-elves in combat, and *Humans* gain more “level points” per level.

2. Combat happens as follows:

   Party ~ A party can consist of up to three characters. They should be chosen from amonst a list of created characters.

   Rounds ~ On each round all alive characters (and their opponents) will take exactly one action. The order of actions is determined at random (but with all elves going before all non-elves).

   Actions ~ Each character can take one of two actions:

   ```
   1. **Attack** a single target of their choice. A dexterity calculation determines their
   2. **Heal** themselves, regaining an amount of lost hit points (but never exceeding the
   ```

Combat End ~ Combat ends when all characters on one side are no longer alive. The side that has won gains experience equal to the "power" of their opponent and split amongst the party members. The opponent's power is computed by allocating 10 points per each level of each character.

```
The losing side gains no experience.
```

Combat AI ~ You may choose to fight an AI-controlled party. You can choose the level of the enemies in the party as well as their number (but no more than 3).

```
The attributes for the enemies created this way should be computed as follows. They all

"Smartness" of enemy AI can be one of three settings:

1. Standard: Only attacks, never heals. Chooses its attack target at random.
2. Resilient: Heals whenever the amount of healing would not result in any lost healing
3. Smart: Heals whenever the amount of healing would not result in any lost healing. Ch
```

Main deliverables

1. You need to provide ways for the user to create new characters, and to form and store parties to use.
2. You need to provide a way for a user to allocate gained level points towards their attributes whenever their character gains a level.
3. You need to provide a system for performing combat.
4. You must offer both player vs player and player vs AI modes.

Optional deliverables

1. Add more race options with interesting special behaviors
2. Add other abilities. For instance:
   - "area of effect" abilities that can target multiple opponents.
   - "friendly heal" instead of self-heal.
3. Create "campaigns" consisting of a specific series of encounters.
4. Create "professions" (healer, caster, warrior etc).

PlaySET

The SET game[2] is a card-matching game. Each card contains 4 characteristics, each with three options:

- Number of items (1, 2, 3)
- Color of the items (red, purple, green)
- Shape of the items (oval, squiggle, rhombus)

[2] http://www.setgame.com/set

- Fill of the items (empty, shaded, solid)

The card deck consists of all 3 * 3 * 3 * 3 = 81 combinations of cards. The goal of the game is to form SETs. A SET is a triple of cards that in each individual characteristic are either all the same (e.g. all red) or all different (i.e. one red, one purple, one green). For any two cards, there is a unique card that complements them to form a SET.

The game is played as follows:

1. The deck is shuffled.
2. 12 cards are placed down in three rows of four.
3. If no SET is present (you will need to write an algorithm to detect that), three more cards are added to the end of each row (if there are cards left). If still no SET is present (an extremely unlikely event), another three cards are added.
4. Depending on skill level (beginner / medium / advanced, you decide on the levels), the player has a given amount of time to find a SET. The timer resets every time the player finds a SET.
5. When a SET is found, the player gains an amount of points depending on the timer left and the skill level (up to you to determine the scoring system). The cards are removed from the board in a properly animated way, and three new cards (if available) are placed in their spot.
6. The game ends when there are no SETs present and no cards remaining on the deck, or when the player runs out of time.
7. If the player runs out of time, they should see a visual of the SET they could not find.

Main deliverables

1. You need to provide a suitable visual for a card. SVG graphics can help you in designing the cards. You also need to arrange a number of cards on a board, and to animate moving cards from the "deck" to the board.
2. You need to implement a process for testing if a given set of cards contains a SET, and which cards form the SET.
3. Users should be able to select/deselect the cards on the board, with some error mechanism showing up if the choices are not valid.
4. You need to provide a way to start a new game, at a certain difficulty level, to show a timer and to compute a score.

Optional deliverables

1. "Training" mode: Once two cards are "selected", the third one that matches them is highlighted if it is on the board, or a visual for "no possible match" is displayed.
2. "Infinite" mode: Once all cards from the deck have been drawn, the already-matched cards are reshuffled and placed back on the deck to be drawn from. At any given time there are always at least 12 cards on the board (15 or 18 if no matches were present).

LinkedListVis

In this project you will create a visualization tool for understanding linked lists. The interface will consist of a SVG window which will represent the list and its elements, followed by a section offering operations you can perform to the list (like adding elements).

Each node should be represented via a rectangle that is vertically cut in two. The left part holds the value at the node, while the right part is the pointer. It should either contain a cross to indicate the null pointer, or it should contain an arrow that points to another node. List elements should be built starting at the top left with 4 elements per row and going back and forth (so right-right-right-right-down-left-left-left-left-down).

Your implementation should be visualizing simple linked lists, with a head that is initially empty and only these forward pointers (i.e. not double-linked lists, no sentinel node). The contained values would be simply strings of text.

The visualizations for operations should be performed at a speed that can be controlled via a UI element. They should by default be slow enough that someone could follow them. They should be accompanied by messages for each step along the way.

The visualization should show every step needed to implement the corresponding operation.

As an example, inserting a new node should do the following, each accompanied by a message:

- Create a new node with empty pointer, and place it lower to where the list currently ends, without any connections to the list yet.
- Check if the head pointer is null, and if it is do the special kind of insertion at the beginning of the list.
- If the head pointer is not null, create a "last" pointer that starts at the first element, and have it follow the next pointers as long as they are not-null, At the end of this process, this "last" pointer should show the last element in the list.
- Add a pointer to that last node, to point to the newly created node.
- Smoothly slide the new node in place (adjusting the pointer as the node moves).

Your implementation should visualize the following operations (this is a lot, discuss with me which subset you would like to implement):

1. Creating a new empty list.
2. Inserting a new node that contains a user-provided string at the beginning.
3. Inserting a new node that contains a user-provided string at the end.
4. Inserting a new node that contains a user-provided string at a specific index.
5. Inserting a new node that contains a user-provided string right after a user-selected (via clicking) node.
6. Removing the node at the beginning.
7. Removing the node at the end.
8. Removing the node at a specific index.

9. Removing a user-selected (via clicking) node.
10. Getting the value of the last node.
11. Getting the value of the first node.
12. Returning the value at a given index.
13. Searching for the first node whose value equals a given string.

Main deliverables

1. A visualization of linked lists as described above, with emphasis on a clear demonstrations of the steps involved for each operation. Try to create something that could then be used as an instruction tool in CS223.
2. An interface supporting the operations described above.
3. A code structured in such a way that the project could somewhat easily be expanded to include other data structures.

Optional deliverables

1. "Cloning/Copying" a list.
2. "Ordered Number List" mode where the entries must be numbers and they are stored in increasing order. You then provide an operation for inserting a number at its proper place, and an operation for testing that the list numbers are properly ordered.
3. "Sorting" operation. Given a list containing numbers, perform a sort on the list by suitably rearranging the nodes, using the above operations. Both selection sort and insertion sort should be reasonably straightforward.
4. Create an option for double-linked lists, with/without a sentinel node.

SortVis

Create a visualization tool for various sorting algorithms (think of it as a simple version of https://visualgo.net/en/sorting).

The arrays to be sorted will contain positive numbers. The main area will contain a window representing the numbers to be sorted, using a user-defined number of vertical bars with heights representing the "size" of numbers.

Main deliverables

1. The visualization should work for all main algorithm methods that rely on swapping elements (insertion, selection, quicksort, bubblesort). At least 3 algorithms should be visualized.
2. The interface should highlight bars when their values are about to be swapped.
3. The interface should highlight bars that are currently "active" in some way (i.e. if a pivot and/or an index is being used, the current elements should be marked somehow). Watching the bars should give someone an indication of how the algorithm works.

4. The interface should keep a count of how many comparisons and how many "swaps" have been performed.
5. The user can ask for a random set of numbers with a given maximum and specific size, or can manually provide a list of numbers in a text area.
6. The user should be able to choose which algorithm to show and the speed at which to show it.

Optional deliverables

1. The user should be able to step through it, performing only one step at a time if they choose to.
2. The system can run a number of simulations for a specific algorithm (or all algorithms) and varying number n of elements, and produce a scatterplot of the number of swaps as a function of n (using different colors if multiple algorithms are used).
3. Multiple algorithms can be seen at the same time, "competing" with each other.

Shapes

Shapes is an app for drawing simple diagrams. It allows you to add elements to a canvas, change their settings, draw connections and so on. Uses SVG graphics to implement.

Main deliverables

1. There is a main canvas with configurable dimensions.
2. The user can pick from a list of "shapes" to add. At the very least, this should include lines, rectangles, ovals, text.
3. Once a shape type is selected (or an actual shape in the graph is selected), the interface below the canvas should offer customization settings (e.g. line type, color).
4. There should be a predefined list of colors.
5. Color/fill/line settings for to-be-created shapes should be remembered from the last shape of that type to be created/edited.
6. A "new" button would allow the shape to be added, to the graph with some initial settings.
7. Elements can be dragged around to be repositioned. Or their coordinates can be directly typed.
8. Dragging starting from a corner should allow resizing (or repositioning of that corner if it is a line).
9. Dragging starting from a size should allow resizing only in that dimension.
10. Lines should have the option to turn either end to an arrow.
11. A list of the created elements is created on the right side. Users can select an existing element by selecting it on that list, or by clicking on it in the canvas.

Optional deliverables

1. Add other shapes (triangles/rhombuses).
2. Allow for curved lines. Control points for these curved lines can be added by double-clicking somewhere along the line.
3. Allow for user-settable colors rather than a fixed list of them. Optionally choose from a pallette.
4. Remember the last 4/5 used colors and offer easier access to them.
5. Allow for automatically connecting two items by an arrow if the user drags from one onto the other.
6. Allow arrows to "lock" onto elements by placing their endpoints near the boundary, and subsequently move with them.
7. Allow for "undoing" any number of steps.

Memory Game

In the memory game a NxN grid of cards is shown, with their content hidden. Cards have some image on their other side, and each image appears on exactly two cards. Users get to click on one card, then another, and if the two cards match then they win this point and the cards are removed from the grid, otherwise the cards are flipped back to have their images hidden.

Main deliverables

1. Creation of a NxN grid, where N should be able to switch between 4, 6 or 8.
2. At least two different sets of available images to use.
3. A timer that keeps count of how long it takes the user to finish.
4. A "top scores" list that shows the best 3 times for each grid size.

Optional deliverables

1. Time restriction after the first card is clicked and before you have to click the second card (or it times out and the first card resets).
2. "Challenge mode", where you see what the cards are only after you have clicked them both (i.e. first card is not revealed the moment you select it).
3. Different patterns for the card backgrounds.
4. Ability for users to upload their own images.
5. "Challenge mode 2", where when two cards don't match then their positions are switched with some other cards in the board (but the user gets to see which positions are swapped).

2048

The 2048 game is played as follows:

1. The board is typically a 4x4 grid. It starts with two random squares "filled" by containing the numbers 2 and 4.
2. Each "round" the user picks a direction, top/bottom/left/right. All filled squares move as far as possible towards that direction, and in addition any squares with the same number that move into each other get combined into one square with twice the number (so when a 4 gets moved into another 4, they get replaced by an 8).
3. After that move happens, a random empty square is filled with a 2. This ends the round.
4. The player wins when he manages a square with number 2048.
5. If it is not possible to fill a square with a 2, the player loses.

Main deliverables

1. A playable grid, where the user clicks on arrows on the four sides to indicate their direction.
2. Squares should get progressively darker colors as the numbers increase.
3. At least 2 different color palettes are available for the user to choose from.
4. The player can reset the game at any time.
5. The moves of the filled squared should be animated.

Optional deliverables

1. Variations of the game with bigger grids and/or different target numbers.
2. The ability for the user to "undo" their last move.
3. A mathematical analysis of the minimum number of moves needed to complete the game.