**CS133J Final Project**

Final Project: JavaScript Game

Due: Sunday June 7, 2020

Points: 100

For this final project, you will use JavaScript to create a simple interactive game that is played in the browser. Some possible games include:

* Classic Arcade Games: games such as Pong), Breakout, Snake, Space Invaders, or Asteroids all involve relatively simple logic (since they were created for early computers!), and thus are simple to re-implement in JavaScript. These games range in difficulty: Pong and Breakout are relatively simple in themselves, while Asteroids can be much more complex. Thus to achieve full credit on this assignment, you may have to include additional features (e.g., "power ups" for Breakout).
* Casual Mobile Games: for a greater challenge, you could implement a simple version of a popular casual or mobile game. For example, Threes have moderately complex and achievable logic, as do the different variations of matching games like Bejeweled. Classic casual games such as Tetris may also be possible.
* Board Game: you could also implement a board game such as Checkers, Othello/Reversi, or Backgammon. Since these are usually two-player games, you could either implement a version were two humans play at the same keyboard (taking turns), or for greater complexity implement a simple "AI" that lets the human play against the computer!
* Other options are also possible: I've seen intro programming students create simple versions of Monopoly, Pokemon, and Super Mario. If you're feeling stuck, you could even implement a version of Memory.

What game you choose is up to you: be sure and read through all of the details described below to make sure that your project will meet the assignment requirements.

Objectives

By completing this final project you will practice and master the following skills:

* Combining HTML, CSS, and JavaScript to create interactive, engaging websites
* Working with fundamental JavaScript functions and data types
* Implementing programs with event-driven logic
* Dynamically manipulating DOM elements and their attributes
* Identifying and utilizing functions from external libraries

Game Requirements

While the exact choice of game (and how it is played) is up to you, it will need to meet a few specific requirements (to make sure you demonstrate your development skills):

Graphical Display

Your game needs to be displayed graphically in the web browser—that is, it should involve pictures and shapes, not just outputted or printed text.

You are welcome to use the HTML5 Canvas (Links to an external site.) to render your game, or you can display and manipulate the DOM elements directory (e.g., creating and positioning styled <div> or <button> elements). You can also create and manipulate SVG elements (e.g., <circle>, <rectangle>) in the same way you manipulate the DOM.

The choice of whether to use Canvas methods or DOM methods is up to you. Games that require the user to interact with or click on specific elements may be easier to implement through the DOM, while games with more complex or fast-moving appearances may work better with the Canvas.

You are allowed to use external libraries to help with the graphical display. For example, Raphael.js provides helpful methods for working with SVG elements. If you want to go all-out, three.js provides a framework for rendering 3D graphics (it's a wrapper around OpenGL, which is used for professional video games).

Many simple games involve animation of some kind: whether a ball moving across the screen in Pong or tiles sliding across the board in Threes. Continuous animations (e.g., a ball moving forever) are most commonly created by repeatedly calling window.requestAnimationFrame() method. More discrete animations (e.g., a tile moving when a user clicks) could be created in the same way (though quickly stopping the callback recursion); could have JavaScript apply CSS animation classes (perhaps from a framework; or could use JavaScript animations from libraries like jQuery.

Start with a simple appearance (rectangles and circles are good), and then add in more complexity and design polish after you get the basic interactions in place!

Note that you will need to create the HTML page to display the game. Your page should include a title and play instructions, as well as information about the current game score (see below). Include some CSS (Bootstrap is fine) to make sure the page looks good!

Interactivity

Your game will need to be interactive and respond to user actions in some way (the player should be able to play it!) Responding to clicks on different DOM elements or at different locations on the Canvas is sufficient, but you can also support using the mouse to click-and-drag or let the user hit keys on the keyboard.

You will need to register event listeners to support this interactivity. As a general approach, have your event callback modify the game state (e.g., what game elements are positioned where), and then re-render the game's display to show the updated state in preparation for the next set of user inputs!

The game's interaction should be pleasant and frictionless. For example, there shouldn't be delays between the user input and the game's response.

Be sure and include instructions on how to play the game—whether to use a mouse or keyboard, and what to click on or type!

Scoring and Timing

Since this is a game, there should be some way to "win" (or at least to achieve a high score before losing!). As such, your game must include some form of scoring system and a way to determine whether that score "wins". Additionally, your game should keep track of the amount of time elapsed that the player has been playing. This can be used as part of the score (e.g., your score can be how long you've kept from losing), or can be an independent value.

The timer should start when the player explicitly begins (either by clicking a "play" button or some other action), and should stop when the player either wins or loses.

Both the current time elapsed and the current score should be displayed to the user. Each value must be displayed as a individual DOM elements (not just printed on a Canvas): for example, you can have a "scoreboard" at the top of the page. You should display the time elapsed in a human-readable format (e.g., 5 seconds), updating regularly (every second is fine). The Moment.js library can help a lot with managing and formatting time.

You could even display the score as a progress bar (Links to an external site.)!

Additionally, when the game is won, you will need to notify the user (and users don't see console.log!). You should do this using a modal window, like the one provided by the Bootstrap framework. (This is not the window.alert() function!). This can just be a small modal with a simple message (like "You won!") and a button that the user can click to "Play Again". Clicking this button should of course restart the game, just as if they had started the game to begin with.

Accessibility

Graphical games are exceedingly difficult to make accessible. There is rarely a textual representation of the game that could be read by a screen reader, and the timing and precision required by games often excludes players with motor impairments.

Nevertheless, it is possible to make sure your game is accessible to people with low to moderate vision impairments (but who do not utilize screen readers). In particular, you should make sure that your game's appearance has the required levels of color contrast (Links to an external site.) (e.g., that there is enough difference between colors so that components can easily be distinguished). You can use a tool such as Colour Contrast Analyzer or this online Colour Contrast Check to help measure the contrast of your chosen colors. In addition, some color palette tools such as the Material Color Picker provide accessibility measures.

Beyond this, you also make sure that the content surrounding the your game (e.g., the instructions, scoreboard, etc) is accessible to screen readers. This means using appropriately semantic HTML and ARIA attributes: use ARIA Live Regions to (politely) cause any dynamically generated text to be read when it appears, and give your "game region" a role of application to make sure that your custom interaction controls do not interfere with screen reader keyboard navigation.

Basically a screen reader should be able to visit your page and navigate through it's content, even if the user cannot effectively play the game!

If you want, you can try to use hidden live regions and other ARIA attributes to also make your game playable by users with screen readers. For example, you might update a live region with state messages ("The ball is moving to the left!" "The top-left tile is now a banana!") that the user can react to via the keyboard. This approach is much more feasible for games that are not timing dependent (Threes is a good candidate). This would be considered sufficient for meeting the "game complexity" requirement (below), and could even earn extra credit depending on the game.

Game Complexity

The game you implement needs to be of sufficient complexity to effectively demonstrate your ability to program with JavaScript. As mentioned above, classic arcade games are often very simple; to earn full credit on this project you will need to add additional complexity/functionality to them. Game complexity can take a couple of forms, including:

Expanded features or capabilities of an existing simple game. For example: "power ups" in Pong or Breakout (e.g., that make the paddle bigger or smaller, produce multiple balls, etc).

Advanced game logic (e.g., the logic of Threes is sufficiently complex).

Additional visual flourishes (e.g., fancy transitions and animations when tiles move or cards flip).

A computer-controlled opponents (AI) with a reasonable level of sophistication (e.g., not just random movement!).

User specified preferences and option, such as different "level" themes or difficulties.

Screen reader support.

A project can demonstrate sufficient complexity by include one of these properties to a great extent (a lot of expanded features OR really fancy animations), or many features to a small extent (a few expanded features AND a simple animation). As a simple heuristic, the following would be worth full credit:

Breakout with 3 different "stages" and 3 different "power-ups"

Threes with sliding cards

Checkers with a computer opponent who makes legal if not-very-effective moves

Memory playable with a screen reader

Structured Code

Even more than HTML and CSS, the style and structure of JavaScript code matters in order to make sure that your code can be understood and improved over time. For this course, good JavaScript programming style means:

Function and variable names should be descriptive, identifying what the function does or what the variable references. Do not use names like stuff, thing, or a as they won't help anyone understand your code. Use verbs for functions and nouns for variables (including plurals where appropriate!) Function and variable names should be in camel-case (e.g., myVariablename).

Constant (unchanging) variables should be declared using const and named in UPPER\_CASE. Hard-coded "magic variables" (size values, etc) should be constants. You can use an Object (e.g., RECT\_SIZE) to "group" related values together into a single namespace.

Prefer let and const over var for declaring variables.

Use proper and consistent indentation. Code inside a block should be indented more than code outside the block.

Remember the semicolons!

Your code should be well-organized into short, reusable functions as much as possible. Try to use functions that do one thing (that is labeled by the function name!), and then compose those functions together into more abstract behaviors.

For example, you might have a renderControlButtons() function that just adds buttons to the DOM, a renderScoreboard() function that just adds the scoreboard to the DOM, and then a renderGame() function that calls the previous two as part of showing everything.

Try to avoid side effects in functions: that is, functions should return data rather than assign a value to a global variable in order to "save" data. Avoid global variables!

In general, declare functions at the top of the script, with statements that call those functions when the script is run at the bottom. Put similar functions next to each other so its easy to find the one you're looking for.

While you should develop and test your code using the console (before showing the results on the screen), these debug messages should be removed or disabled before submitting your work. Production code should never litter the console with debug messages—it looks sloppy. A good practice is to wrap console.log() with your own function and use a variable to quickly enable/disable debug logging.

Similarly, be sure and remove any significant pieces of code not used in the production version, so it doesn't get in the way of understanding. Do not submit large blocks of old commented code if that code will never be used again!

Your code should be documented (commented) enough so that other developers can understand it. There is no required documentation format you must follow, but your comments should be intelligible to other developers. Comments should define information that is NOT present in the code itself. For example, use a comment to explain what types of variables a function takes as parameters or returns, and use a comment to describe the motivation and purpose behind a particular algorithm. You can also use comments to help "organize" the code, grouping different types of functions together.

At the very least, include a short comment above each function indicating what it does.

It's OK to utilize small snippets of code that you find on the web, but make sure you include a comment with a URL to the source. Also adjust the identifiers so they make sense in context.

The goal of "good programming style" is to make it so that the code can be read and understood, and so that it is easy to modify the code for further use later. "Poor style" is making coding decisions that get in the way of understandability and modifiability.

**Development Advice**

This final project is purposefully open-ended, as what program you implement is largely up to you. However, here is some general guidance on how you can go about developing your project:

1. First, figure out what game you want to make! Make sure you're familiar with the game, and come up with a vision for what you want it to look like and how the user will play it. Being clear on what you're trying to achieve is the first step of software development.
2. Determine what information (variables) will be needed to represent the **state** of the game at any given moment in time. For example, you may need to keep track of the position of the ball and paddles in Pong, or what tiles are displayed in *Threes*. Define an Object to keep track of this state (you may want to assign "initial" values to it with a setupGame() function).
   * This state object may contain other nested objects. For example, there may be a gameState.ball.xPosition value.
   * Having a second "mid-game" state object to use for testing can also help.
3. Determine how to **display** the game based on the current values of the *state*. Define a render() function (or multiple functions!) that takes in a state object and uses those values to show the game on the screen either by painting on a Canvas or manipulating DOM elements. You should be able to call this render() function with the initial state and see the entire game.
4. Add interactivity to your game. Start by registering listeners to respond to user events (just have them log out a message to check that you're hearing events). Then modify the listener callbacks so that the game state changes based on the events that occur! For example: a listener for the arrow key might change the position of the paddle stored in the game state.

After changing the game state, you should *re-render* the game's display: this will cause it to "react" to the user (and have things like paddles seem to move)! You can do something similar with a timer/animation callback: change the ball's state to move it, then *re-render* the display based on the updated state.

These callbacks are where the complex "game logic" good: what changes when the user clicks on a particular tile or in a specific location? After moving the ball to animate it, how do you decide if it hit a paddle?

See also [Anatomy of a video game (Links to an external site.)](https://developer.mozilla.org/en-US/docs/Games/Anatomy) for a description of this structure (though I would avoid mirroring their example code).

Thinking through the logic and structure of each of these steps can be tricky. Do not hesitate to reach out for help for your particular project!

**Submitting Your Project**

To submit your project, do the following:

1. Confirm that the project is complete and meets all the requirements. A player should be able to start a game on your page, play it to completion, and then start another.
2. A journal of tracking your work from start to finish. Detailed explanation of what approach you use to start the project, what work or did not work, challenges, changes and reflection on the project. I want you to take me on this journey with you. This should be in a word document.

Important Dates:

1. Sunday May 17 Submit Project Title and Plan in Moodle.
2. Sunday May 24 Submit Draft of Work.
3. Monday June 1 Present to Instructor and other students. Will scheduled during class time.

Rubric

**JavaScript Game Rubric**

| JavaScript Game Rubric | | |
| --- | --- | --- |
| **Criteria** | **Ratings** | **Pts** |
| This criterion is linked to a Learning Outcome Graphical Display  Your game is graphically displayed in the browser. The graphical view of the game is robust and styled. You have included surrounding information such as the game title, instructions, and controls. |  | 15.0 pts |
| This criterion is linked to a Learning Outcome Interactivity  Your game is interactive and playable. Users can interact with game objects through mouse or keyboard events. Game play is functional and frictionless. |  | 20.0 pts |
| This criterion is linked to a Learning Outcome Scoring and Timing  Your game tracks its current score and time elapsed, and displays this information to the user. The program determines when the game is won, and shows the player a modal message (that allows them to play again). |  | 15.0 pts |
| This criterion is linked to a Learning Outcome Accessibility  Your game has accessible levels of color contrast. The HTML page itself is accessible and using ARIA attributes to support dynamically generated content. |  | 10.0 pts |
| This criterion is linked to a Learning Outcome Game Complexity  Your game is of sufficient complexity to show off your awesome programming skills! This rubric item is scored on game complexity beyond minimal interactivity. |  | 20.0 pts |
| This criterion is linked to a Learning OutcomeCode Structure and Style  Your code has good style and structure to make it readable and modifiable. Variables and functions are effectively and informatively named. The use of global variables is minimized. Code is organized into functions. Loops and data structures are use appropriately. You have included sufficient comments, including on each function declaration. |  | 20.0 pts |
| Total Points: 100.0 | | |