CSC324 GCIEL Team

GCIEL Assessment Strategy Update: Technical Documentation

Spring CSC324-02: Software Design and Development

Authors: Kelton Watts, Ian Brown, Josh Sutton, Ethan Yuen, Haobo Chen
5-8-2024

Table of Contents

User Stories

User Story 1: ARCS Model-Based Evaluation

As an educational researcher, I want to use an ARCS model-based questionnaire to evaluate the effectiveness of the VR game in terms of attention, relevance, confidence, and satisfaction.

Acceptance Criteria:

- Questionnaire Accessibility
    - The app must provide a dedicated section/tab for the ARCS model-based questionnaire.
    - Users should be able to access the questionnaire easily from the main interface.
- Comprehensive ARCS Model Coverage
    - The questionnaire should comprehensively cover the four components of the ARCS model: attention, relevance, confidence, and satisfaction.
    - Each section of the questionnaire should contain relevant questions that accurately assess its respective ARCS component.
- Ease of Use and Clarity
    - The questionnaire should be easy to understand and user-friendly, with clear instructions and question wording.
    - The layout and design of the questionnaire should facilitate a smooth user experience without unnecessary complexity.
- Data Privacy and Security
    - The app must ensure the privacy and security of the responses collected through the questionnaire.
    - Users should be informed about how their data will be used and stored.

User Story 2: Data Uploading

As a user, I want to upload my dataset in CSV format so that I can analyze my data using the app's features. I want to visualize my data without prior knowledge of how to visualize data so I can focus on the game and my data gathered from the game experience.

Acceptance Criteria:

- The app must allow the uploading of CSV files.

- It should validate the format and structure of the uploaded CSV file.
- The app should provide feedback if the uploaded file does not match the expected format.
- The data visualizations provide information on how to understand the visualization.

User Story 3: Data Analysis

As a data analyst, I want to use various analytical features to analyze my dataset to extract meaningful insights about the game.

Acceptance Criteria:

- The app should offer features like Total Completion Time per Piece, 2D Distance Analysis, Video Engagement Analysis, and Player Positions Heatmap.
- Each feature must accurately process and visualize the data. The app should handle large datasets efficiently.

User Story 4: Data Export

As a researcher, I want to download the available data descriptions and datasets so that I can use them for further analysis or reporting.

Acceptance Criteria:

- The app should provide options to download data descriptions and datasets in CSV format.
- Downloaded files should maintain data integrity and format.

Development Process

Data

With the addition of real data, the app versions can be split into two where ideal mock data is used and where real data is used. First data is converted from a text file into a csv through R code creating playerID, X, Y, and Z positions of player and current piece being placed, and what the player is currently looking at within the experience. The X and Z positions require mutations shifting the X by 5.3 and the Z by -7.4 to get correct positions due to differences in data collection camera position and player position. With this new data set mutations are required to gain completion times and distance from piece pick-up. The mutations occur with a copy of the input data set when they are selected in the app.

Player Location Animation 1 Player Select Menu

The menu for Player Location Animation 1 is data specific. The menu gives the user options to select which players will be displayed in the animation based on the data specified. The menu utilizes the renderUI function of R Shiny to accomplish the reactive menu portion after the data has been entered by the user. The values for the menu are all unique values of the playerID column. Without the reactive menu specification of the players to visualize would not be reproducible with other data sets.

Architectural Design

The previous layout of our app had a simple tab system for each visualization and the survey. The left side had a section for inputting the data file that was always present to the viewer. The data input did not always need to be present, and the data table and survey tabs did not fit in as visualizations in the same way the other tabs did. To fix this we created a bar at the top to separate each function: data input, data visualization, and survey.

The data table was moved to the data input section to help the user ensure they selected the correct data. Within the data visualization section, the tab system was revised to simplify the layout of the visualizations. The tab section received subtabs that change with the selection of a parent tab. This allows the user to select visualizations based on the purpose. The left side panel updates with each visualization to provide context and potential customization options for each visualization. The survey was simply moved to its own section to place a higher focus on its presence and make filling the survey out easier due to the larger share of the screen.

The change to the app's organization adds complexity to a previous simple app but allows for future scaling as more visualizations are needed. The top bar allows other sections/functions to be added to the app. The tab system allows for more visualizations to be added as the need arises. The left side panel allows for more context to be written about the visualizations and what the user may want to pay attention to. Customization by the user can also be added as necessary in this side panel. Placing this information below would not be as clear as the distinct side panel, which helps to replace complexity.

Design Decisions

Description (what-why-how)

Player Piece Viewing

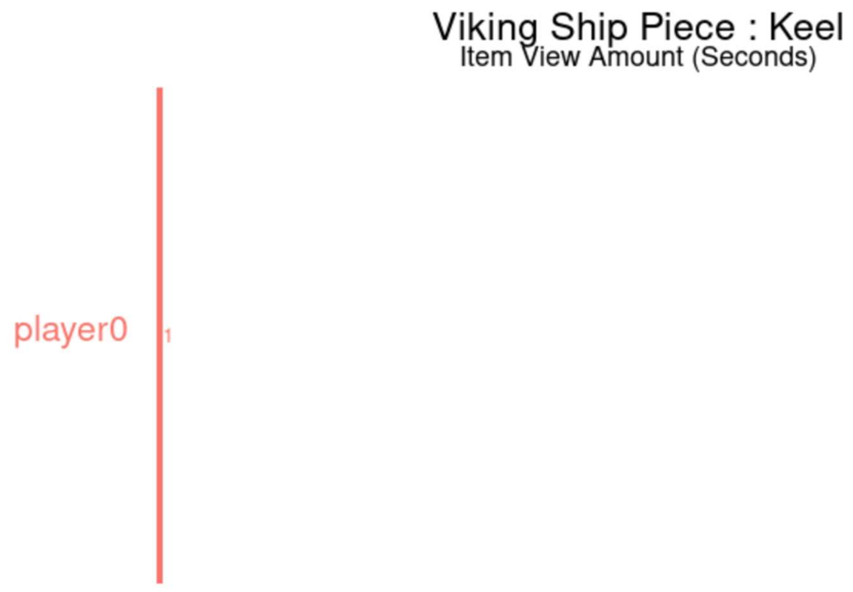


Figure 1: Player Piece Viewing Example. Time spent viewing the Keel piece by player0.

What: The player piece viewing visualization displays all players and the amount of time spent viewing each piece model in seconds.

Why: With these visualizations it gives progression of time spent between each piece and ultimately shows how engaged each player is throughout the experience.

How: Through geom_tiles a bar chart can be made using the piece_look_time values as input for height of each tile. The animation flows through each piece viewed while keeping playerIDs at stagnant positions along the y-axis.

Player Location GIF Animation

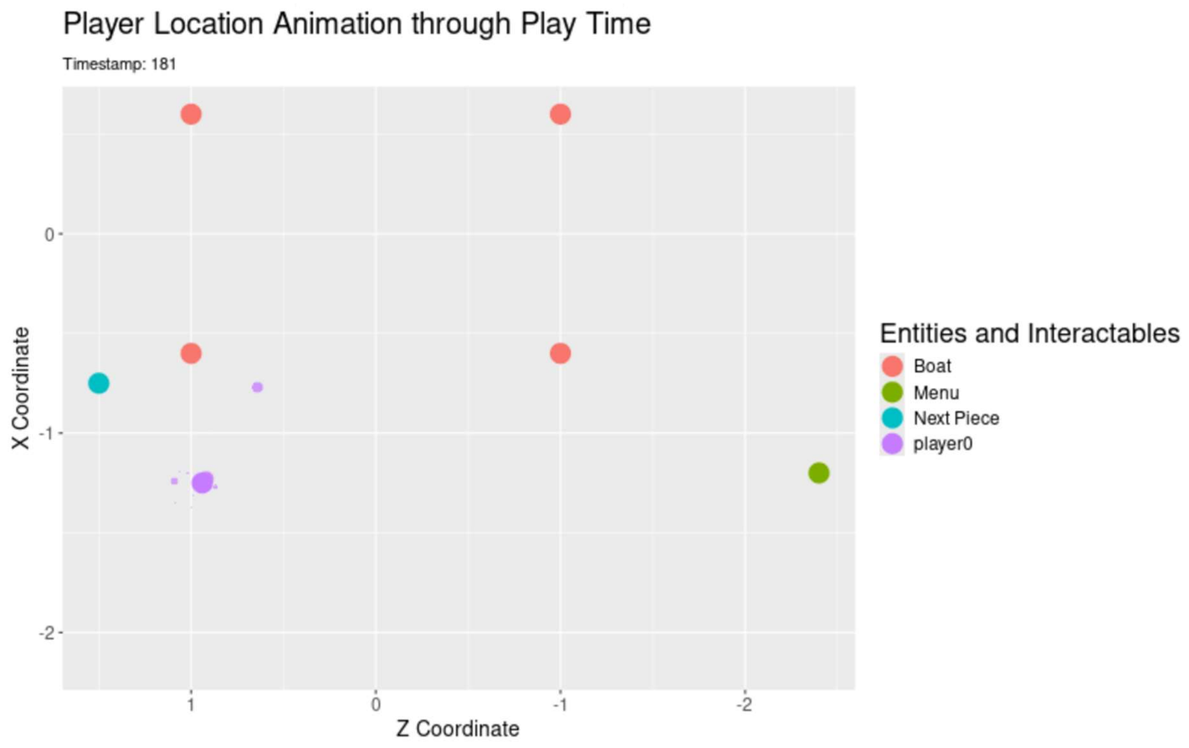


Figure 2: Screenshot Example Player Location Animation for player0.

What: The visualization is a player position over time point graph. The graph includes points for experience interactable objects including the Viking ship, next piece location, and the boat height menu.

Why: This visualization quickly recaps the players experience and through these, new developers can use this visualization to get a better understanding of where players struggled or spent more time learning about each piece.

How: Through gg_animate each geom_point of the player position along the X and Z axis are displayed. This gives a bird's eye view of the where the player was facing the same direction as where the player starts the experience.

Player Location Video Animation

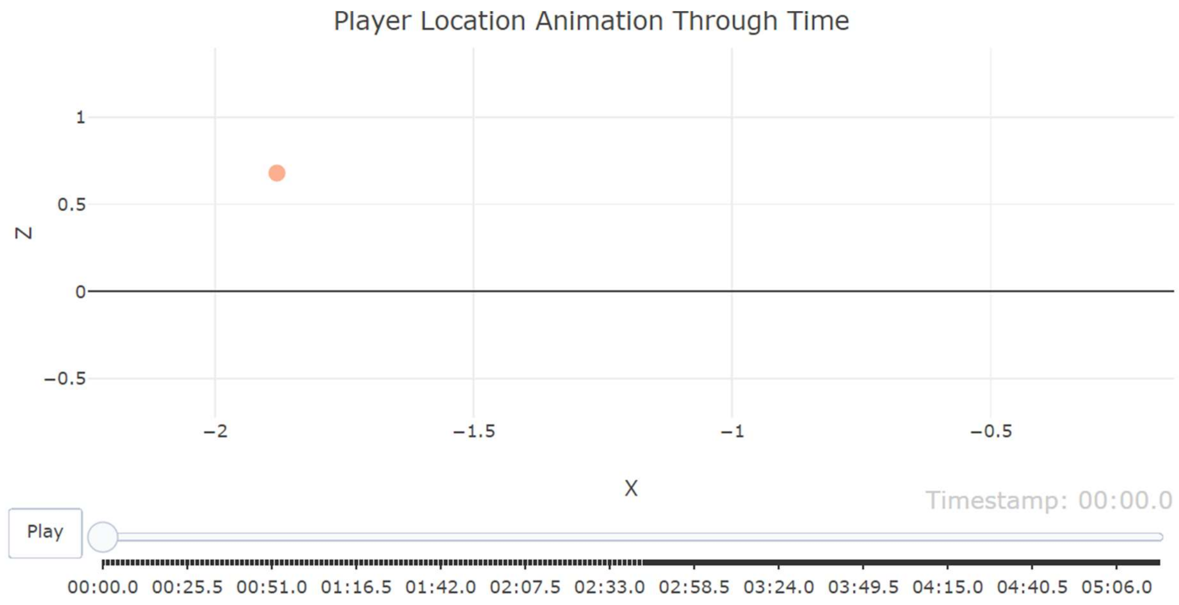


Figure 3: Screenshot Example of Player Location Video Animation

What: A scatter plot of all player positions over time with utilizing the Timestamp field to create each frame of the video.

Why: With a similar purpose to the GIF animation this animation also allows the user to pause the animation where they please to give more time to observation to a certain time in a play through.

How: Through plot_ly creates a scatter plot of player position along the X and Z axis. To specify the animation needs animation_opts is used to create the number of frames and how the scatter plot will animate between frames.

UML Diagrams

UML Use Case Diagram: Assessment Strategy App

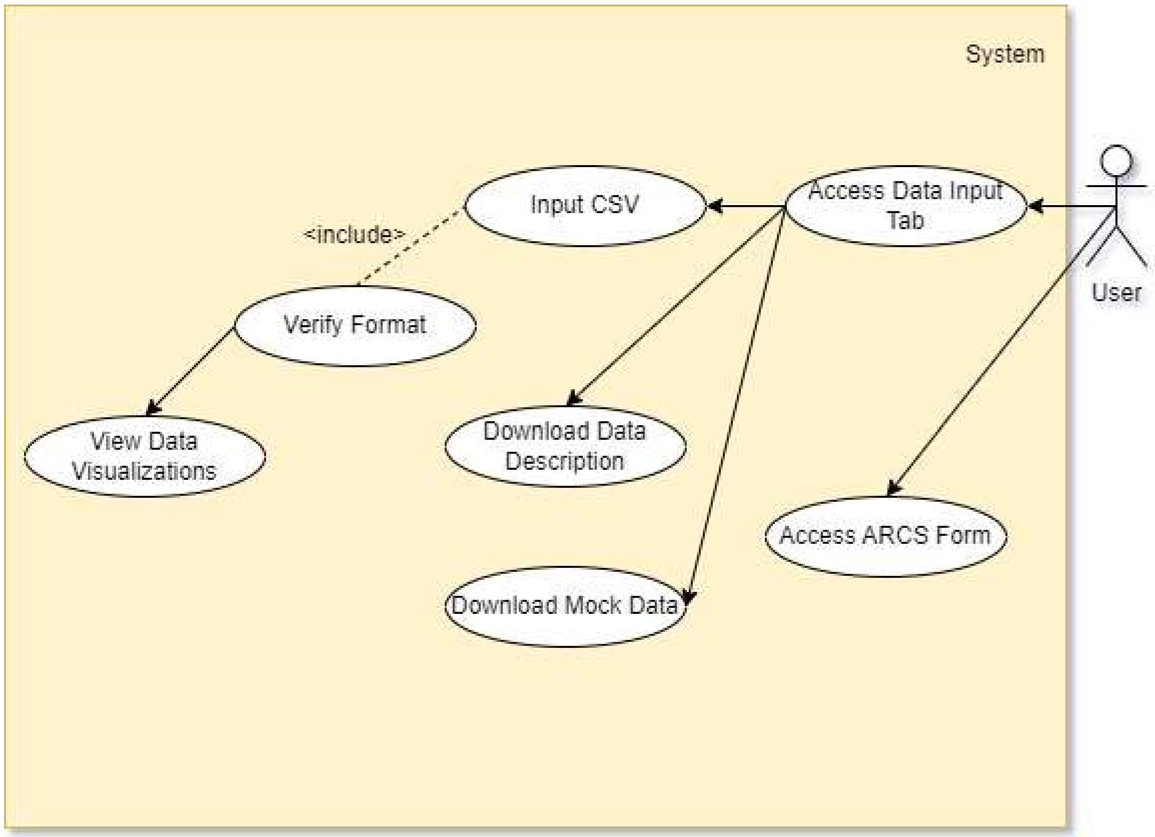


Figure 4: UML Use Case Diagram of Updated Assessment Strategy App

This use case diagram goes over the 4 specified user stories found above. The app must be broken up into data input and the ARCS google form for feedback. This will allow users to understand they can't see visualizations until they input a data set. Within the data input tab, the user can see how the data should be formatted through mock data and what each field means within data description.

UML Activity Diagram: Application Selectability

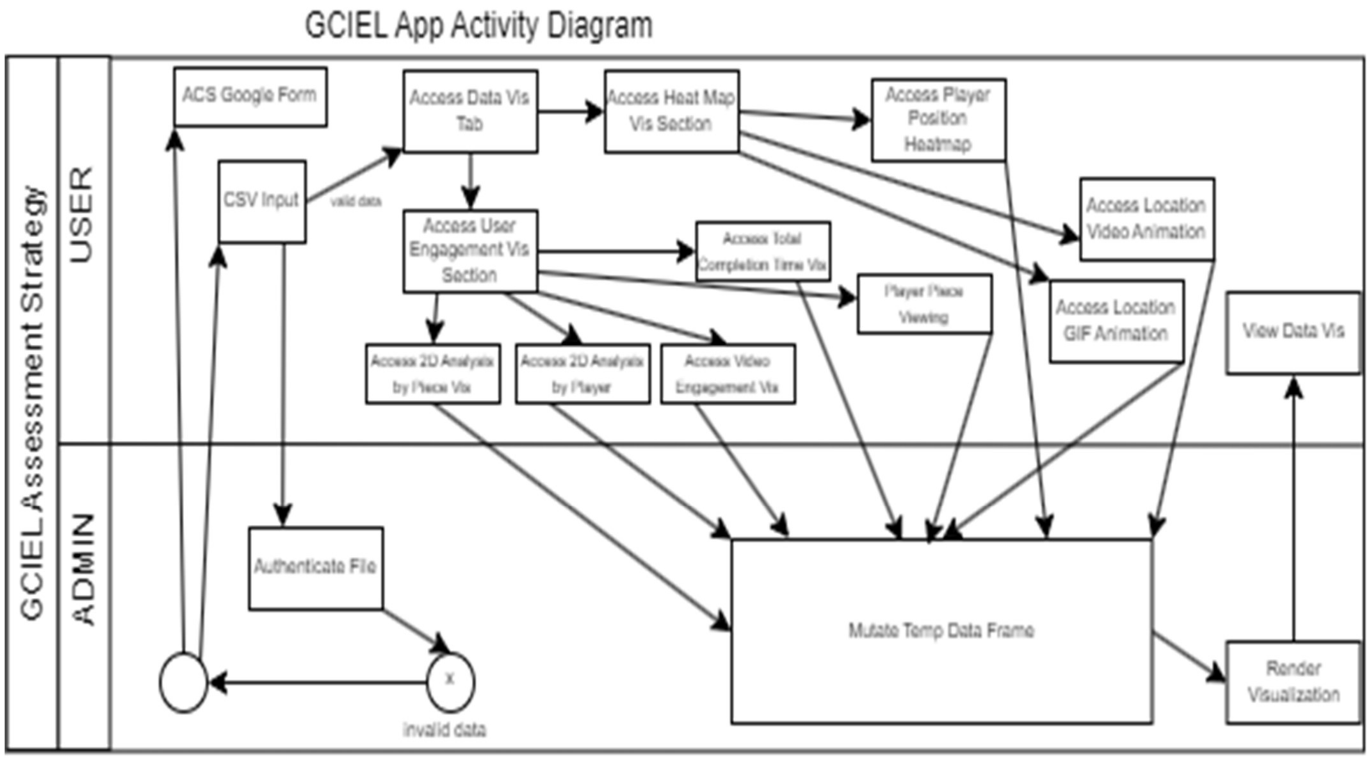

GCIEL App Activity Diagram
GCIEL Assessment Strategy
USER
ADMIN
ACS Google Form
CSV Input
valid data
Access Data Vis Tab
Access Heat Map Vis Section
Access Player Position Heatmap
Access Location Video Animation
Access User Engagement Vis Section
Access Total Completion Time Vis
Player Piece Viewing
Access Location GIF Animation
View Data Vis
Access 2D Analysis by Piece Vis
Access 2D Analysis by Player
Access Video Engagement Vis
Authenticate File
x
invalid data
Mutate Temp Data Frame
Render Visualization

Figure 5: UML Activity Diagram of Updated Assessment Strategy App

The UML Activity Diagram on the Assessment Strategy app is a design on what functionalities the app can perform and what the app does to allow each step to occur. Upon opening the app, the user can either input a csv file or begin the ARCS Google form. When an invalid file csv file is entered, the app will not continue to give access to data visualizations. When a valid data set with correct fields is entered the user is given the opportunity to access data visualizations. With each visualization a temporary data frame is made to summarize and mutate the data to generate the appropriate data to create the graph or animation. We include each selectable piece of the data visualization within this UML diagram.

UML Sequence Diagram: Future Vision

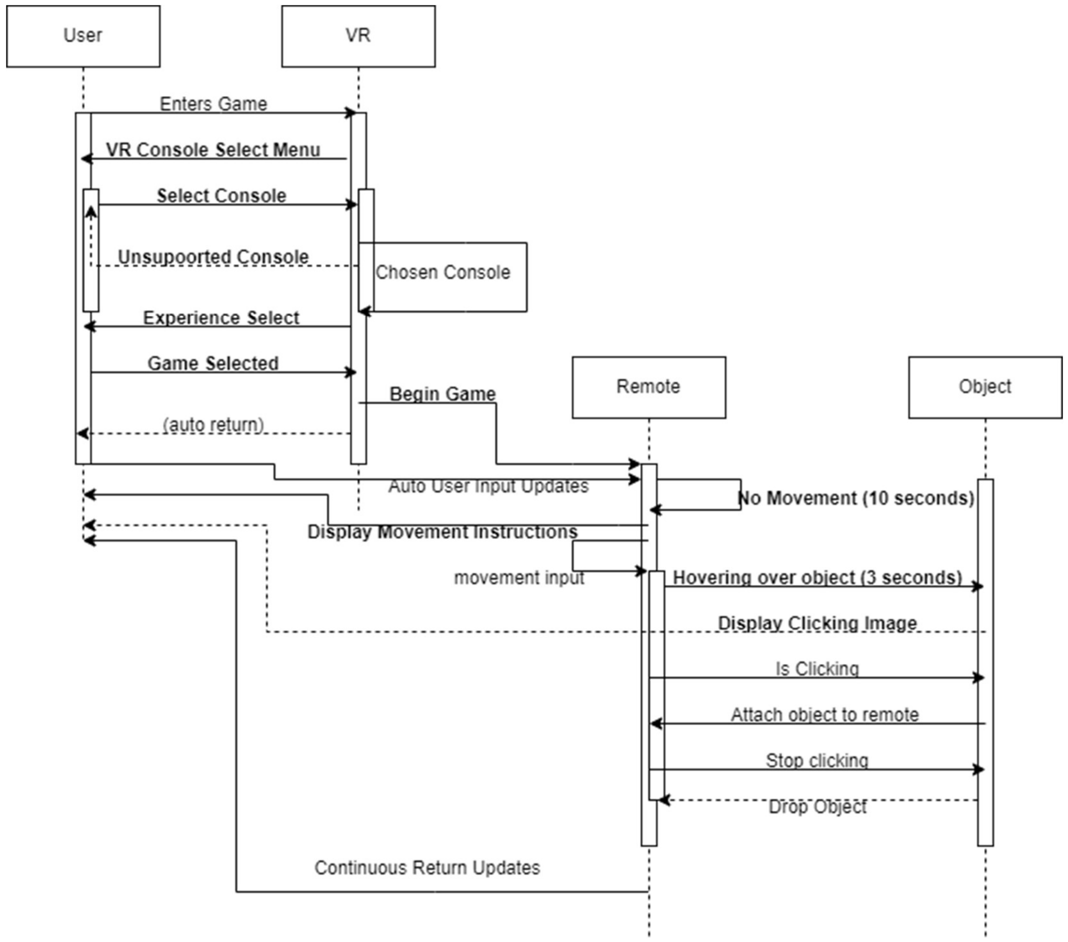


Figure 6: UML Sequence Diagram of Future Vision proposed updates.

The UML Sequence Diagram for Future Vision is a design based on the current state of the virtual reality experience with the added implementations of the Future Vision. As can be seen, the framework is split into four phases: the user, the VR, the remote controller, and the objects. Of course, these are not all aspects of VR, rather just the ones that relate to the proposed changes. The first changes come between the user and the VR with the console select interface and the game selection. Then the game moves between the user and the controller. Here there are proposed built-in safeguards added which account for the lack of movement of a player resulting in a pop-up of instructions, as well as a user not clicking on an object, which results in more pop-up instructions. All of this is added in and amongst the already implemented user interface.

UML Activity Diagram: Future Vision

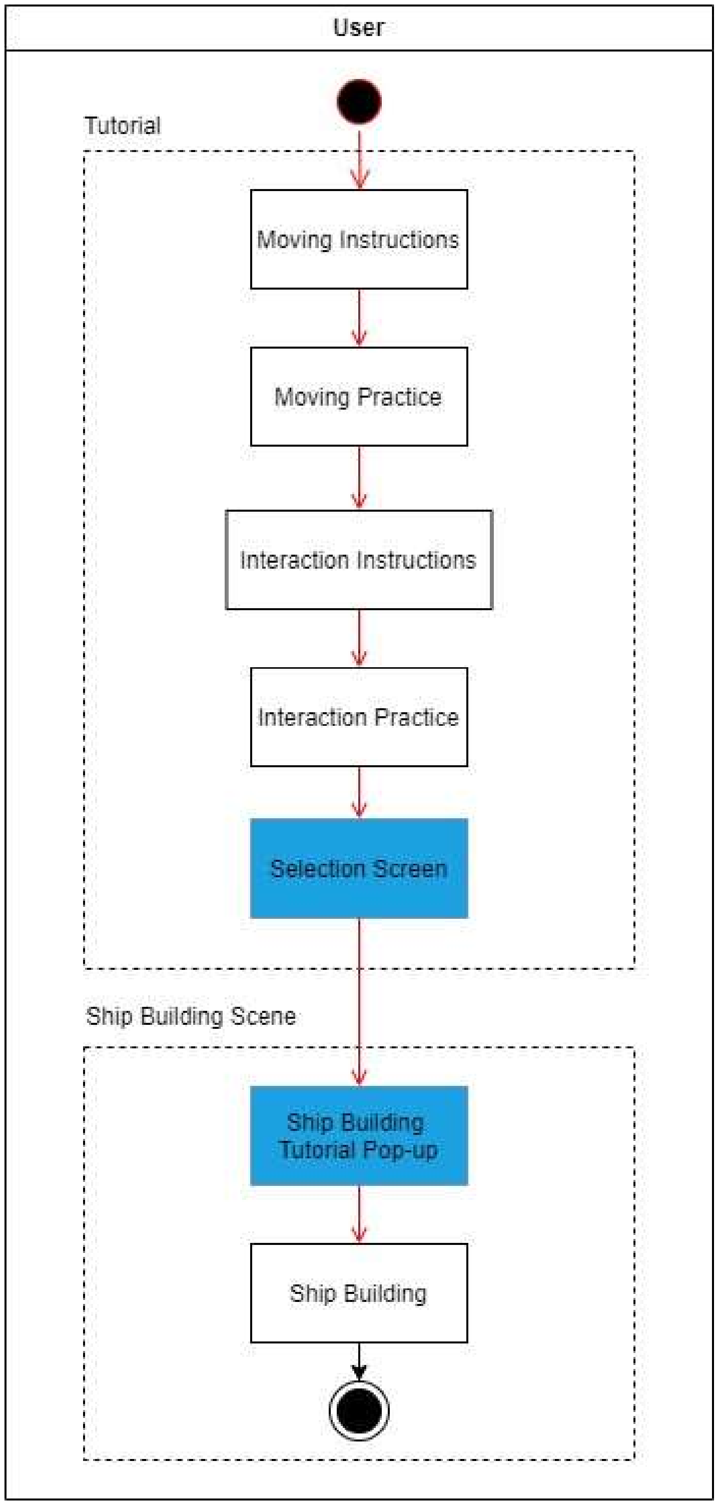


Figure 7: UML Activity Diagram of Future Vision proposed updates.

The Activity Diagram of the ship building scene is quite simple. The addition of the selection screen and ship building tutorial pop-up would allow for potential scalability with other scenes and an improved ship building experience respectively. The additions are in blue. See the Future Vision document for more information.

UML Use Case Diagram: Future Vision

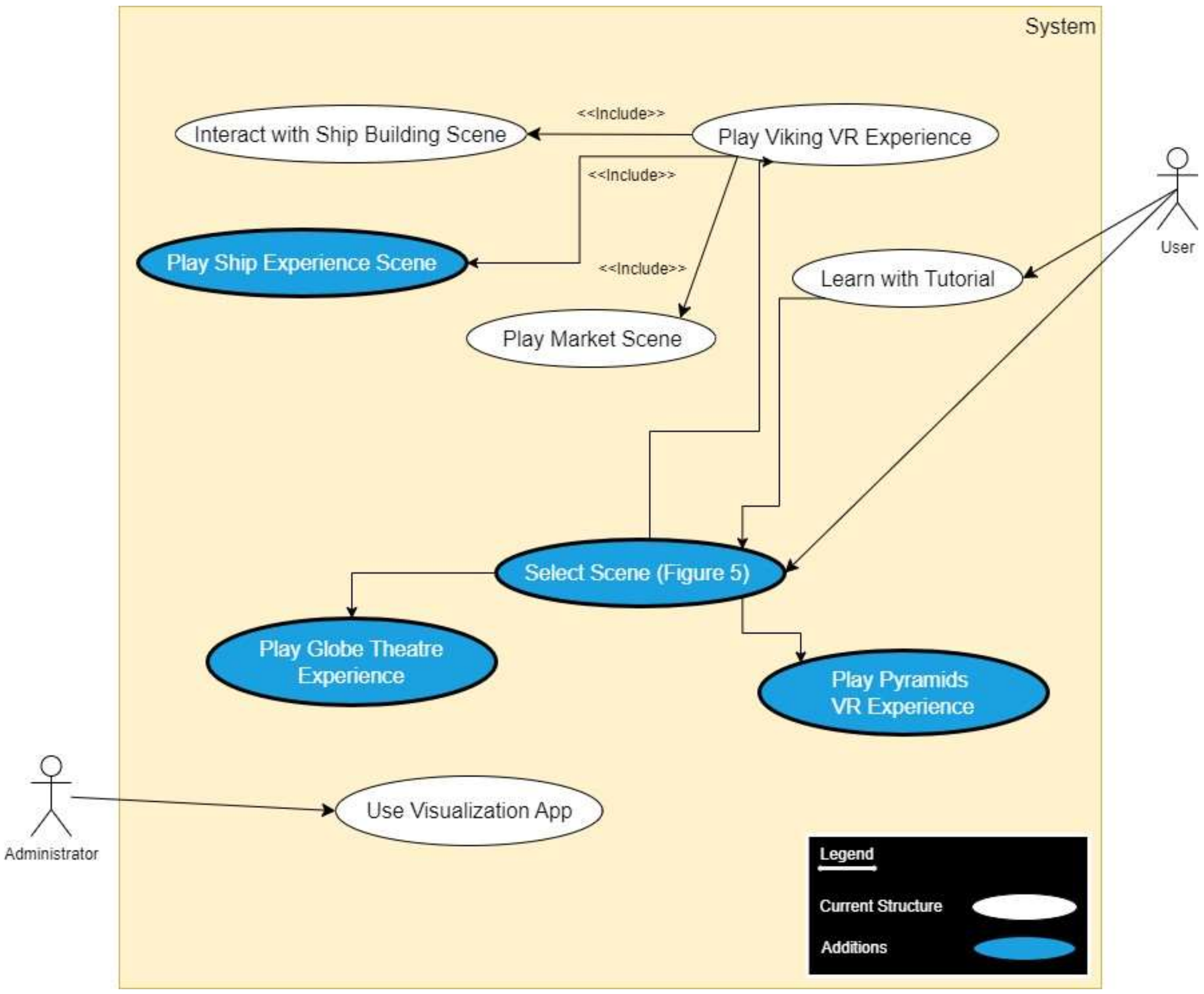


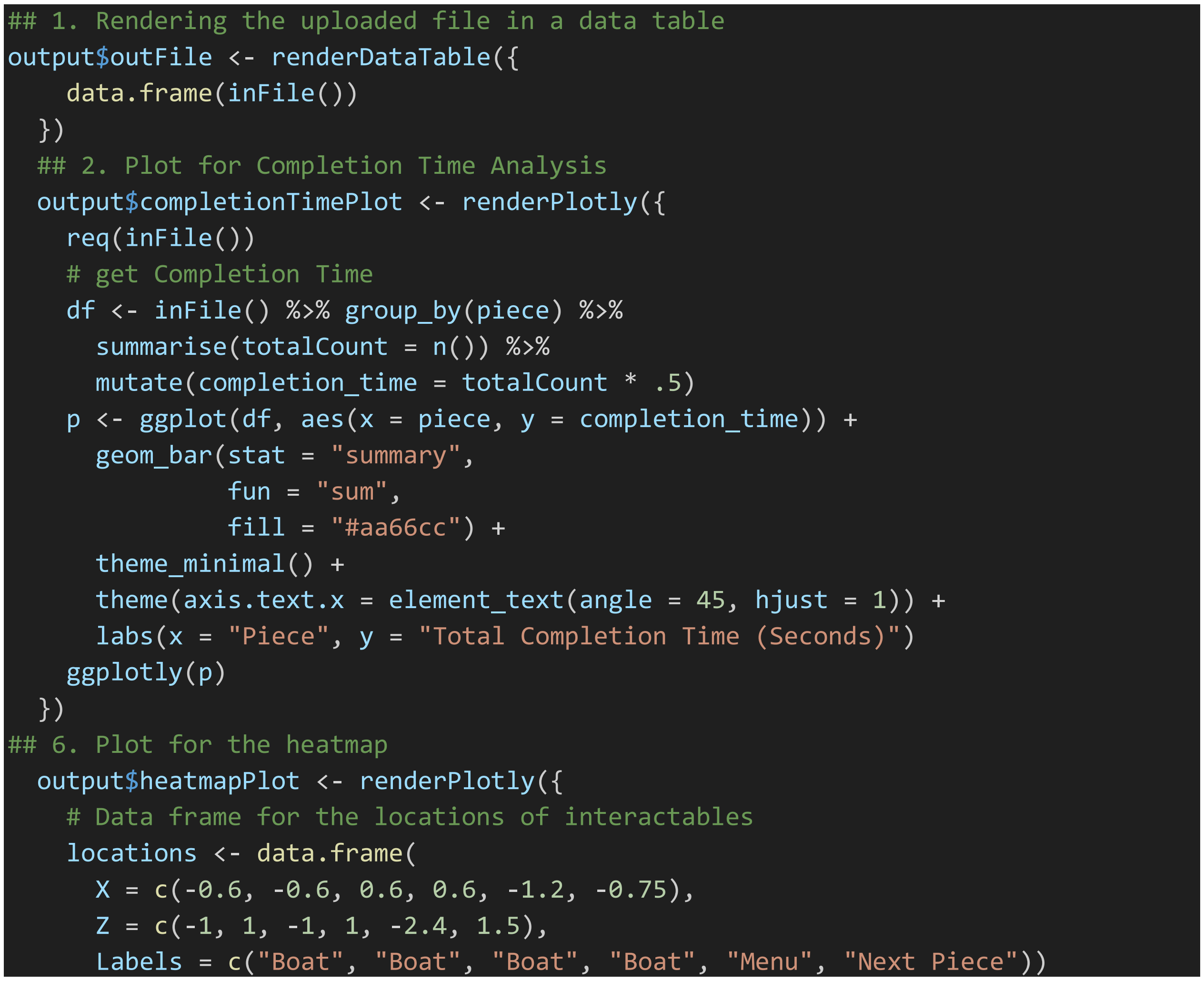
Figure 8: UML Use Case Diagram of Future Vision proposed updates.

The use case diagram above is for the GCIEL VR experience. The addition of the screen select would prepare for scaling into other VR experiences such as the Glove Theatre or Pyramids VR experience (shown in blue). The ship experience scene would be a great addition to the Viking VR experience to allow users to better understand the importance of the ship. See the Future Vision document for more information.

Design Patterns

In our UI/UX design decision we opted to use ShinyUI's fluidPage layout for our visualizations. Due to this, the incorporation of design patterns became difficult. However, the *Singleton Pattern* fits our needs perfectly. The singleton pattern is a creational design pattern which deals with how objects are created and relieves the need to instantiate these objects directly. This pattern ensures that there is only one instance of a class in the program. With a plethora of different visualizations, data sets, and data presentation, this is certainly the case in our application. Between the UI and the Server, there are no two similar visualizations. Therefore, if needed, the code could be split up by visualization and presented in different classes. This is not done in our code; however, it was deemed unneeded by the team, and still meets the criteria of the singleton design pattern. Therefore, our code is presented as the singular continuation of a program.

Example Code: Use of Singleton Pattern Through Different Viz Blocks



```r
## 1. Rendering the uploaded file in a data table
output$outFile <- renderDataTable({
    data.frame(inFile())
  })
  ## 2. Plot for Completion Time Analysis
  output$completionTimePlot <- renderPlotly({
    req(inFile())
    # get Completion Time
    df <- inFile() %>% group_by(piece) %>%
      summarise(totalCount = n()) %>%
      mutate(completion_time = totalCount * .5)
    p <- ggplot(df, aes(x = piece, y = completion_time)) +
      geom_bar(stat = "summary",
               fun = "sum",
               fill = "#aa66cc") +
      theme_minimal() +
      theme(axis.text.x = element_text(angle = 45, hjust = 1)) +
      labs(x = "Piece", y = "Total Completion Time (Seconds)")
    ggplotly(p)
  })
## 6. Plot for the heatmap
  output$heatmapPlot <- renderPlotly({
    # Data frame for the locations of interactables
    locations <- data.frame(
      X = c(-0.6, -0.6, 0.6, 0.6, -1.2, -0.75),
      Z = c(-1, 1, -1, 1, -2.4, 1.5),
      Labels = c("Boat", "Boat", "Boat", "Boat", "Menu", "Next Piece"))
```

Figure 9: Example Code of a Singleton used in the Assessment Strategy App.

Code Construction Guidelines

Functions, Methods, and Size:

- Functions and methods should be small and perform a singular function.
    - This allows repeated calls of functions, avoiding repeated code.
- If functions are too large, refactor them such that they're smaller.
- Smaller functions/methods require fewer tests as they are more straightforward.

Formatting, Layout, and Style:

- Use consistent indentation to define code block. An indentation is created by two spaces per indentation level instead of tab key.
- Include space around all operators (=, +, -, <-, etc.). Ex. x <- 2 * y
- Maintain the length of line below 100 characters and break lines to improve readability.

White Space:

- Empty lines to separate code blocks (i.e., group functions/methods together that serve the same intention)
- Make sure all statements within a block are equally aligned/indented and avoid double indentations.
- Avoid using irrelevant spaces around operators and functions/methods.

Block and Statement Style Guidelines:

- One statement per line, and format statements blocks identically (i.e., if opening bracket is on the same line as the function call [if {], then maintain that throughout
- Utilize more parentheses than necessary around functions.
- With conditionals, put each condition on its own line.

Declaration Style Guidelines:

- Use only one declaration per line. However, you can group related variables together.
- Declare variables close to where they are used. (declaration before use rule).
- Order declarations sensibly: Group your declarations by types and usage.
- Use white space to separate your declarations.
- No nested headers and no source code in headers.

Commenting Style Guidelines:

- All comments should be identified with # followed by a space.
- Comments for a new function should preceded with two blank lines.
- For functions, use comments to provide the description of the function's purpose, parameters, and return value.
- Use TODO comments to indicate incomplete areas of your code that require further work or attention.

Identifier Naming Conventions:

- Useful and comprehensible variable and function names
- Camel case variables and functions when applicable ex. shipDist
- Use verb-noun pair for function names ex. getDistance()

File and Folder Naming Conventions:

- File names will be capitalized.
- Files and folders will be named by [Owner][Modifier][Object] format.
    - Owner: The owner should be specified for individual notes or drafts.
    - Modifier: If a specific file or folder differs from another file or folder of the same objects in some way the modifier should specify the distinction.
        - Ex: unrevised, real, mock, etc.
    - Object: Each file or folder should be described by the type of object the file or folder represents.
        - Ex: Coding Guidelines, app, Data Visualizations, etc.

Defensive Programming:

- Need to protect from bad data.
- Need to Validate!
- On data file upload:
    - Check file operations: Did the file open? Did the read operation return anything? Did the write operation write anything? Did we reach EOF yet?
- Always initialize variables and don't depend on the system to do the initialization for you.

Error Handling:

- To avoid error during publication, keep consistent error handing practices with well-rounded tests.
- Recover:

- Recovery means that your program needs to try to ignore the bad data, fix it, or substitute something else that’s valid for the bad data.

GIT Commit Conventions:

- Branch names will contain the main purpose of the branch in no more than 3 words.
- Branches created will have “-“ to separate words.
- Commits will be in the format of [Verb][Object].
    - Ex: Fixed player animation visualization, created interactive menu, etc.
- Commit messages will also contain the location of the update.
-

Note: The GIT Commit Conventions were added after and will be for the future (this is a revision based on class feedback).

References

1. JEF Works. (n.d.). R Style Guide. Retrieved from https://jef.works/R-style-guide/ Accessed 04/19/2024.

2. Dooley, J. F. (2017). Software Development, Design and Coding With Patterns, Debugging, Unit Testing, and Refactoring (2nd ed. 2017.). Apress. https://doi.org/10.1007/978-1-4842-3153-1