

Module Interface Specification for
SFWRENG 4G06:
Dice Duels: Duel of the Eights

Team 9, dice_devs

John Popovici

Nigel Moses

Naishan Guo

Hemraj Bhatt

Isaac Giles

January 16, 2025

1 Revision History

Table 1: Revision History

Date	Developer(s)	Change
25-01-10	Hemraj Bhatt	Added content to section 3
25-01-11	John Popovici	Added content to section 4; Updated section 2
25-01-14	Naishan Guo	Added content to section 6
25-01-14	Nigel Moses	Added content to section 5
25-01-15	Naishan Guo	Updated content to section 6
...

Contents

1	Revision History	i
2	Symbols, Abbreviations and Acronyms	1
3	Introduction	1
4	Notation	2
5	Module Decomposition	3
5.1	Hardware-Hiding Modules	3
5.2	Behavior-Hiding Modules	3
5.3	Software Decision Modules	3
6	MIS of [Module Name —SS]	4
6.1	Module	4
6.2	Uses	4
6.3	Syntax	4
6.3.1	Exported Constants	4
6.3.2	Exported Access Programs	4
6.4	Semantics	4
6.4.1	State Variables	4
6.4.2	Environment Variables	4
6.4.3	Assumptions	4
6.4.4	Access Routine Semantics	4
6.4.5	Local Functions	5
7	Appendix	28

2 Symbols, Abbreviations and Acronyms

See [SRS Documentation](#) and [MG Documentation](#).

3 Introduction

The following document details the Module Interface Specifications for the Duel of the Eights game. The game takes inspiration from Yahtzee and introduces a platform that enables players to create custom Yahtzee-like game variants. This platform includes preset options such as classic Yahtzee and an octahedron version, while also offering flexibility for users to define their own game variables. Players can customize aspects such as the number and type of dice, ranging from cubed (6-sided) to octahedral (8-sided) and other multi-sided dice, as well as scoring mechanisms. Scoring can either follow the traditional end-of-game calculation seen in classic Yahtzee or adopt a per-round format, allowing for head-to-head matchups. The game can be played both locally and online, allowing players to be able to play with one another regardless of their distance from one another.

Complementary documents include the System Requirement Specifications and Module Guide. The full documentation and implementation can be found at <https://github.com/John-Popovici/duel-of-the-eights/tree/main>.

4 Notation

The structure of the MIS for modules comes from [Hoffman and Strooper \(1995\)](#), with the addition that template modules have been adapted from [Ghezzi et al. \(2003\)](#). Template used from [the SFWRENG 4G06GitHub](#).

The following table summarizes the primary scene-building classes and elements used by *Dice Duels: Duel of the Eights* through Godot. Many different types of nodes are used, each with specific functions and properties.

Object	Description
Object	Dynamic base class holding properties, methods, signals, and scripts.
Node	Inherits object, can be assigned as a child of another node.
Scene	A tree of nodes.

The following table summarizes the primary coding data types used by *Dice Duels: Duel of the Eights* through Godot.

Data Type	Notation	Description
null	<i>null</i>	Denotes the absence of a value.
boolean	<i>bool</i>	Holds one of 2 values, true or false.
integer	<i>int</i>	A number without a fractional component within the bounds of signed 64-bit integer type $[-2^{63}, 2^{63} - 1]$.
float or decimal	<i>float</i>	A decimal number, as representable by 64-bit double-precision floating-point.
string	<i>String</i> or " <i>...</i> "	A sequence of unicode characters.
signal	<i>signal</i>	Allows connected objects to react to events.

The following table summarizes the primary data structures used by *Dice Duels: Duel of the Eights* through Godot.

Structure	Example	Description
array	[5, " <i>Hello</i> ", 2.3]	Sequence of data elements.
dictionary	{" <i>White</i> " : 35, " <i>Red</i> " : 10}	Set of key and value pairs.

The specification of *Dice Duels: Duel of the Eights* uses some derived data types such as Vector3. Vector3 contains three floating point numbers, representing coordinates or direction vectors. In addition, *Dice Duels: Duel of the Eights* uses functions, which are defined by the data types of their inputs and outputs. Local functions are described by giving their type signature followed by their specification.

5 Module Decomposition

The following section is taken directly from the Module Guide document for this project.

The modules are categorized into three types: Hardware-Hiding, Behavior-Hiding, and Software Decision modules. Below is the hierarchy:

5.1 Hardware-Hiding Modules

- **NetworkManager2P Module:** Manages connection and synchronization for two-player games.
- **GameUI Module:** Provides the interface for interacting with the game and displaying relevant data.

5.2 Behavior-Hiding Modules

- **MultiGameManager Module:** Manages the sequence of multiple Yahtzee games and customization phases.
- **PlayerManager Module:** Tracks player states, scores, and upgrades.
- **GameManager Module:** Handles a single game of Yahtzee.
- **GameSettings Module:** Loads and stores settings for this Yahtzee variant.
- **CustomizationMenu Module:** Implements dice and game customization between games.
- **DynamicScoreboard Module:** Tracks and displays scores dynamically.
- **CustomBaseDie Module:** Handles the 3D dice models, textures, and physics.
- **DynamicDiceContainer Module:** Manages dice interactions and rendering.

5.3 Software Decision Modules

- **ScoreCalculator Module:** Calculates scores for dice rolls based on Yahtzee rules and custom modifiers.

6 MIS of [Module Name —SS]

[Use labels for cross-referencing —SS]

[You can reference SRS labels, such as R??. —SS]

[It is also possible to use L^AT_EX for hyperlinks to external documents. —SS]

6.1 Module

[Short name for the module —SS]

6.2 Uses

6.3 Syntax

6.3.1 Exported Constants

6.3.2 Exported Access Programs

Name	In	Out	Exceptions
[accessProg —SS]	-	-	-

6.4 Semantics

6.4.1 State Variables

[Not all modules will have state variables. State variables give the module a memory. —SS]

6.4.2 Environment Variables

[This section is not necessary for all modules. Its purpose is to capture when the module has external interaction with the environment, such as for a device driver, screen interface, keyboard, file, etc. —SS]

6.4.3 Assumptions

[Try to minimize assumptions and anticipate programmer errors via exceptions, but for practical purposes assumptions are sometimes appropriate. —SS]

6.4.4 Access Routine Semantics

[accessProg —SS]():

- transition: [if appropriate —SS]
- output: [if appropriate —SS]

- exception: [if appropriate —SS]

[A module without environment variables or state variables is unlikely to have a state transition. In this case a state transition can only occur if the module is changing the state of another module. —SS]

[Modules rarely have both a transition and an output. In most cases you will have one or the other. —SS]

6.4.5 Local Functions

[As appropriate —SS] [These functions are for the purpose of specification. They are not necessarily something that is going to be implemented explicitly. Even if they are implemented, they are not exported; they only have local scope. —SS]

References

- Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall, Upper Saddle River, NJ, USA, 2nd edition, 2003.
- Daniel M. Hoffman and Paul A. Strooper. *Software Design, Automated Testing, and Maintenance: A Practical Approach*. International Thomson Computer Press, New York, NY, USA, 1995. URL <http://citeseer.ist.psu.edu/428727.html>.

This is the draft MIS section

3.1 MIS of MultiGameManager Module

3.1.1 Module

MultiGameManager

3.1.2 Uses

GameSettings, GameManager, CustomizationMenu, networkmanager2p

3.1.3 Syntax

3.1.3.1 Exported Constants

None.

3.1.3.2 Exported Access Programs

- `finish_game(winner: String, myPlayerFinalStats: Dictionary, OpponentFinalStats: Dictionary)`
- `loadGameSetup()`

3.1.4 Semantics

3.1.4.1 State Variables

- `current_game_count`: Tracks the number of completed games
- `total_games`: Total number of games in the sequence.
- `self_wins`: Tracks the number of wins the user has
- `opponent_wins`: Tracks number of wins the opponent has
- `game_settings`: Tracks the current settings and parameters for the current game
- `hand_settings`: Tracks the settings and requirements for the valid dice hands of the current game session (i.e. requirements for a full house, requirements for a yatzee, etc.)

3.1.4.2 Environment Variables

N/A

3.1.4.3 Assumptions

All dependencies, such as GameSettings and GameManager are initialized.

3.1.4.4 Access Routine Semantics

- `finish_game(winner: String, myPlayerFinalStats: Dictionary, OpponentFinalStats: Dictionary)`
Transition: Updates win count, triggers customization phase.
- `loadGameSetup()`
Transition: Retrieves and implements the game settings for this session.

3.1.4.5 Local Functions

- `start_next_game()`
Description: Determines whether to begin a new game or if it's time to end the multi-game session
- `start_game()`
Description: Initializes the game settings and starts a new game.
- `end_game()`
Description: ends the entire multi-game session and announces the winner.
- `_on_settings_ready(_game_settings: Dictionary, _hand_settings: Dictionary)`
Description: Confirms the game and hand settings selected by the user for the game, and then informs the other online players of the selected settings for the game.
- `recieve_game_settings(_game_settings: Dictionary, _hand_settings: Dictionary)`
Description: recieves the game and hand settings from the host player, and uses these settings when starting a new game.

3.2 MIS of GameManager Module

3.2.1 Module

GameManager

3.2.2 Uses

NetworkManager2P, PlayerManager, ScoreCalculator, DynamicDiceContainer, DynamicScoreboard, GameUI

3.2.3 Syntax

3.2.3.1 Exported Constants

None.

3.2.3.2 Exported Access Programs

- `_on_settings_ready(_game_settings: Dictionary, _hand_settings: Dictionary)`

3.2.4 Semantics

3.2.4.1 State Variables

- `current_round`: Tracks the current round in the ongoing game.

- `roll_count`: Tracks the current roll count in the ongoing round.
- `game_over`: Tracks if the game has ended.
- `round_start_done`: Tracks if the start of each round has been completed. Used for Multi-player synchronization.
- `rolls_read`: Tracks if the rolls have been read. Used for Multi-player synchronization.
- `start_next_round`: Tracks if the next round of each round can be started. Used for Multi-player synchronization.
- `rematch`: Tracks if a rematch can be started. Used for Multi-player synchronization.
- `roll_selection_done`: Tracks if the roll selection has been completed. Used for Multi-player synchronization.
- `roll_selection`: Tracks the players roll selection choice (Roll or Pass).
- `hand_selection_done`: Tracks if the hand selection has been completed. Used for Multi-player synchronization.
- `start_next_round`: Tracks the hand selected to score.

3.2.4.2 Environment Variables

Network manager for multiplayer actions.

3.2.4.3 Assumptions

Game rules are preloaded, and dice configurations are valid.

3.2.4.4 Access Routine Semantics

- `_on_settings_ready(_game_settings: Dictionary, _hand_settings: Dictionary)`
Transition: Receives settings to set up the game and start.

3.2.4.5 Local Functions

- `setup_game() -> void`
Description: Calls other related setup and game initialization functions then starts the game
- `start_round() -> void`
Description: Starts each round
- `rollPhase(roll: bool) -> void`
Description: Triggers the effect of the chose roll phase (roll/pass)
- `recieveRolls(fromHost: bool, _rolls: Array[int]) -> void`
Description: Receives the values of the dice to continue the game flow

- `setup_selection() -> void`
Description: Sets up the roll selection options
- `recieveRollSelection(_type: String) -> void`
Description: Receives Roll Selection to continue the game flow
- `recieveHand(hand: Dictionary) -> void`
Description: Receives had selection to trigger scoring
- `recieveBonus(hand: Dictionary) -> void`
Description: Receives Bonus if applicable from hand for scoring
- `endOfRoundEffects() -> void`
Description: Triggers end of round effects
- `endGame() -> void`
Description: Triggers End Game effects
- `restartGame() -> void`
Description: Triggers the start of a new game
- `exitGame() -> void`
Description: Exits game and returns to Intro Scene
- `_on_game_state_recieved(state: String, data: Dictionary) -> void`
Description: Triggers effects of receiving game state from connect peer device
- `setup_PlayerManager(settings: Dictionary) -> void`
Description: Initializes Player Manager for self and enemy player
- `setup_scoreboard() -> void`
Description: Sets up Scoreboard with hands available
- `setDisableAllButtons(state: bool) -> void`
Description: Used to toggle the disabled state of all buttons
- `setDisableScoreBoardButtons(state: bool) -> void`
Description: Used to toggle the disabled state of the scoreboard buttons
- `setDisableRollButtons(state: bool) -> void`
Description: Used to toggle the disabled state of the roll buttons (Roll/ Pass)
- `_on_roll_selected() -> void`
Description: Triggers the effects of selecting the roll button
- `_on_pass_roll() -> void`
Description: Triggers the effects of selecting the pass button

- `waiting_on_other_player(isWaiting: bool) -> void`
Description: Used to Toggle the visibility of the waiting for other player overlay
- `_on_hand_selected(hand: Dictionary) -> void`
Description: Triggers the effects of selecting a hand
- `_on_bonus_exist(_hand: Dictionary) -> void`
Description: Triggers the effects of a bonus against the current hand
- `setup_game_environment(_game_settings: Dictionary) -> void`
Description: Sets up the game environment (3D elements)
- `returnToIntro() -> void`
Description: Triggers the effects of returning to Intro Scene

3.3 MIS of PlayerManager Module

3.3.1 Module

PlayerManager

3.3.2 Uses

DynamicDiceContainer, networkmanager2p

3.3.3 Syntax

3.3.3.1 Exported Constants

- N/A

3.3.3.2 Exported Access Programs

- `setup_player(myPlayer: bool, initial_health: int, _playerName: String, _hostDevice: bool, _dice_container: Node3D)`
- `update_score(new_score: int)`
- `get_total_score()`
- `get_Last_score()`
- `adjust_health(points: int)`
- `getHealth()`
- `roll_dice()`
- `roll_selected_dice()`
- `pass_roll()`

- `setRolls(_rolls: Array)`
- `getRolls()`
- `get_dice()`
- `clearRolls()`
- `getStats()`
- `getState()`

3.3.4 Semantics

3.3.4.1 State Variables

- **score:** Contains the players current total score accumulated throughout the current game.
- **last_score:** Contains the most recent score achieved by the player during the last round.
- **health_points:** Contains the current health of the player.
- **rolls:** Tracks the players current dice roll values
- **dice:** Tracks the players physical dice.
- **selected_hand:** Tracks the players chosen dice hand for the current round
- **myPlayer:** checks if this instance of playermanager is for the player themselves, or for the player's opponent
- **playerName:** Name of the player
- **hostDevice:** checks if player's device is the current host of the game

3.3.4.2 Environment Variables

Input from game events or player actions.

3.3.4.3 Assumptions

Each player has a unique `player_id`.

3.3.4.4 Access Routine Semantics

- `setup_player(myPlayer: bool, initial_health: int, _playerName: String, _hostDevice: bool, _dice_container: Node3D)`
Transition: Initialized the player with their default values and game settings.

- `update_score(new_score: int)`
Transition: updates the total score of the player to include the new score from the most recent round, and updates the last score to be the new score achieved during that round.
- `get_total_score()`
Output: Returns the player's total score.
- `get_Last_score()`
Output: Returns the player's last score from the previous round.
- `adjust_health(points: int)`
Transition: updates the health of the player.
- `getHealth()`
Transition: Returns the player's health.
- `roll_dice()`
Transition: Rolls all the Dice. The rolled dice are tracked, and the new dice roll values retrieved.
- `roll_selected_dice()`
Transition: Rolls only the dice specified by the player. The rolled dice are tracked, and the new dice roll values retrieved.
- `pass_roll()`
Transition: Skips the dice roll for that turn
- `setRolls(rolls: Array)`
Transition: Sets the values of the player's dice to the specified value
- `getRolls()`
Output: Returns the dice values
- `get_dice()`
Output: tells the requester which dice belongs to the player
- `clearRolls()`
Transition: clears the dice from the board and resets the dice values.
- `getStats()`
output: Returns a few of the players state variables, providing the player's basic stats for the game (Name, total score, current health, etc)
- `getState()`
output: Returns ALL of the players state variables tracked by this module, providing all relevant information about the player for the game.

3.3.4.5 Local Functions

- `checkIfDiceValidThenRead()`

Description: This function checks the results of dice rolls to determine if they are valid. If there are invalid or indeterminable die results, then all the invalid dice will be re-rolled. When all the dice produce valid results, it will allow the results to be read by the system.

- `roll_rolling_or_invalid_dice()`

Description: re-rolls any dice that is either still rolling, or has an invalid result that makes it's value unreadable due to any reason, such as rolling out of bounds for example.

- `readRolls()`

Description: Reads the values of each die that has been rolled and tracks the results

3.4 MIS of NetworkManager2P Module

3.4.1 Module

NetworkManager2P

3.4.2 Uses

None.

3.4.3 Syntax

3.4.3.1 Exported Constants

None.

3.4.3.2 Exported Access Programs

- `start_server(_port: int)`
- `connect_to_server(_hash: String)`
- `getIsHost() -> bool`
- `getHashIP() -> String`
- `getHashPort() -> String`
- `disconnect_from_server()`
- `broadcast_game_state(state: String, data: Dictionary)`
- `send_game_settings(_game_settings: Dictionary, _hand_settings: Dictionary)`

3.4.4 Semantics

3.4.4.1 State Variables

None.

3.4.4.2 Environment Variables

None.

3.4.4.3 Assumptions

Assumes a reliable network transport layer. Players do not exceed the 2-player limit for this module.

3.4.4.4 Access Routine Semantics

- `start_server(_port: int)`
Transition: Establishes a server on the host device for other player to connect to.
- `connect_to_server(_hash: String)`
Transition: Attempts to establish a connection to a host server to join a game.
- `getIsHost() -> bool`
Transition: Returns if the device is the host or client of the game.
- `getHashIP() -> String`
Output: Returns the Hash code for communication to other connecting player.
- `getHashPort() -> String`
Transition: Returns the Hash code for the port for communication to the other connecting player.
- `disconnect_from_server()`
Transition: Disconnects from the host server and cleans state.
- `broadcast_game_state(state: String, data: Dictionary)`
Transition: Broadcasts the game state to the other player for synchronization.
- `send_game_settings(game_settings: Dictionary, hand_settings: Dictionary)`
Transition: Broadcasts the game settings to the other player to start the game.

3.4.4.5 Local Functions

- `_on_peer_connected(id: int) -> void`
Description: Triggers the effects of a peer connection
- `_on_peer_disconnected(id: int) -> void`
Description: Triggers the effects of a peer disconnection
- `_on_server_disconnected() -> void`
Description: Triggers the effects of the server disconnection
- `_on_connection_failed() -> void`
Description: Triggers the effects of a failed connection
- `_try_reconnect() -> void`
Description: Triggers attempting to reconnect to other player

- `numberToLetters(number: int) -> String`
Description: Helper function of the ip to hash function
- `lettersToNumber(letters: String) -> int`
Description: Helper function for the has to ip function
- `ip_to_hash(ip: String) -> String`
Description: Returns a the hashed IP for easy communication of IP
- `hash_to_ip(hash_code: String) -> String`
Description: Returns the IP from its Hashed version
- `broadcast_disconnect(state: String, data: Dictionary)`
Description: Broadcasts disconnection to other player on forced disconnect
- `recieve_disconnect()`
Description: Receives disconnection from other player
- `recieve_game_state(state: String, data: Dictionary) -> bool`
Description: Receives game state from other player to transmit to listening modules
- `ping() -> void`
Description: Sends a ping (triggered on a regular basis) to check status of connection
- `recieve_game_settings(_game_settings: Dictionary, _hand_settings: Dictionary)`
Description: Receives game settings from host player

3.5 MIS of ScoreCalculator Module

3.5.1 Module

ScoreCalculator

3.5.2 Uses

N/A

3.5.3 Syntax

3.5.3.1 Exported Constants

N/A

3.5.3.2 Exported Access Programs

- `initializeValues()`
- `calculate_hand_score() -> Array[int]`

3.5.4 Semantics

3.5.4.1 State Variables

- **SinglesTotal:** Variable used to track the total score of every "singles" type hand for the purposes of bonus score calculation
- **BonusThreshold:** Tracks the maximum bonus score that can be given to the player's score for that round
- **BonusScore:** The bonus score added in addition to the players normal score acquired for that round.
- **BonusUsed:** Checks if the bonus score has been applied yet.
- **BonusExists:** Checks if a bonus score should be applied for this round

3.5.4.2 Environment Variables

N/A

3.5.4.3 Assumptions

The dice rolls are valid, and that no invalid hand type is requested

3.5.4.4 Access Routine Semantics

- **initializeValues()**
Transition: Initialized the default values for the state variables of this module
- **calculate_hand_score() -> Array[int]**
Output: detects the type of hand. Then, based on the hand type, it calculates the score before returning the player's score for the round.

3.5.4.5 Local Functions

- **setupBonus(_bonusHand: Dictionary)**
Description: This function initializes the variables and parameters relevant to calculating a bonus score value. This function is used when a bonus score needs to be added to the regular score for a hand.

3.6 MIS of DynamicScoreboard Module

3.6.1 Module

DynamicScoreboard

3.6.2 Uses

N/A

3.6.3 Syntax

3.6.3.1 Exported Constants

N/A

3.6.3.2 Exported Access Programs

- `populate_scoreboard(hand_settings: Dictionary)`
- `updateButtonScore(_score: int)`
- `updateBonusButtonScore(_score: int)`
- `setAllButtonsDisable(state: bool)`

3.6.4 Semantics

3.6.4.1 State Variables

- `lastButton`: Tracks the last button pressed by the player
- `BonusButton`: Tracks the bonus button
- `AllButtons`: Tracks every single button relevant to the hands and their corresponding scores on the scoreboard.
- `hand_selected`: tracks the currently selected hand
- `bonusExists`: Checks if a bonus exists for the current hand
- `TotalLabel`: Tracks the total overall score for the player.

3.6.4.2 Environment Variables

- `vbox_container`: used to help interact with the UI interface to help display the scoreboard.

3.6.4.3 Assumptions

Assumes that every hand option to be displayed is unique to each other and valid for the rules specified by the game settings

3.6.4.4 Access Routine Semantics

- `populate_scoreboard(hand_settings: Dictionary)`
Output: Populates the scoreboard with valid hands allowed by the game settings
- `updateButtonScore(_score: int)`
Transition: Updates the score for a selected hand
- `updateBonusButtonScore(_score: int)`
Transition: Updates the bonus score that will be added to the score for a hand

- `setAllButtonsDisable(state: bool)`
Transition: enables or disables all the buttons relevant to scoring for the scoreboard.

3.6.4.5 Local Functions

- `_on_hand_selected(hand: Dictionary, button: Button)`
Description: A callback function that detects when a player selects a hand, and then updates the rest of the module about the selection.

3.7 MIS of CustomizationMenu Module

3.7.1 Module

CustomizationMenu

3.7.2 Uses

N/A

3.7.3 Syntax

3.7.3.1 Exported Constants

N/A

3.7.3.2 Exported Access Programs

- `show_customization()`
- `wait_customization()`

3.7.4 Semantics

3.7.4.1 State Variables

N/A

3.7.4.2 Environment Variables

N/A

3.7.4.3 Assumptions

Assumes that a full round has been properly completed.

3.7.4.4 Access Routine Semantics

- `show_customization()`
Output: Shows the customization options available to the player and allows them to select customization options in between games
- `wait_customization()`
Output: Waits for the opponent to select customization options in between games.

3.7.4.5 Local Functions

- `pass_customization()`

Description: Skips the customization phase for the player

3.8 MIS of DynamicDiceContainer Module

3.8.1 Module

DynamicDiceContainer

3.8.2 Uses

CustomBaseDie

3.8.3 Syntax

3.8.3.1 Exported Constants

N/A

3.8.3.2 Exported Access Programs

- `roll_dice()`
- `roll_selected_dice()`
- `roll_rolling_or_invalid_dice()`
- `get_dice_values()` -> `Array[int]`
- `get_dice()` -> `Array[RigidBody3D]`
- `get_selected_dice()` -> `Array`
- `get_rolling_dice()` -> `Array`
- `get_invalid_dice()` -> `Array`
- `add_dice(dice_count: int, dice_type: int)`
- `move_dice_aside()`
- `move_dice_in_line()`

3.8.4 Semantics

3.8.4.1 State Variables

- `dice_nodes`: Tracks each dice object that is a part of this container module.
- `start_positions`: Specifies the starting positions of the dice

3.8.4.2 Environment Variables

N/A

3.8.4.3 Assumptions

Assumes all dice are of the same type.

3.8.4.4 Access Routine Semantics

- `roll_dice()`
Transition: Rolls all the dice with a random force and torque
- `roll_selected_dice()`
Transition: Rolls only the dices specified by the player.
- `roll_rolling_or_invalid_dice()`
Transition: Rerolls any dice identified as an invalid roll or trapped in an endless roll.
- `get_dice_values() -> Array[int]`
Output: retrieves the value of each rolled dice
- `get_dice() -> Array[RigidBody3D]`
Output: retrieves each dice object that is a part of the container
- `get_selected_dice() -> Array`
output: retrieves a list of each die selected by the player
- `get_rolling_dice() -> Array`
output: gets a list of all dice that is currently rolling
- `get_invalid_dice() -> Array`
Output: gets a list of all dice that have been rendered invalid, whether due to its improper landing position, unreadable value, improper graphic rendering, or other issue.
- `add_dice(dice_count: int, dice_type: int)`
Transformation: Adds new dice objects to the container
- `move_dice_aside()`
Transformation: Moves dice not being re-rolled to the side of the board to prevent the dice from interfering with the reroll
- `move_dice_in_line()`
Transformation: Move dice once to an organized display on the board to make them easier to read.

3.8.4.5 Local Functions

- `clear_dice()`
Description: removes each dice object that's part of the container.

3.9 MIS of CustomBaseDie Module

3.9.1 Module

CustomBaseDie

3.9.2 Uses

N/A

3.9.3 Syntax

3.9.3.1 Exported Constants

N/A

3.9.3.2 Exported Access Programs

- `roll()`
- `get_face_value() -> int`
- `set_dice_ui(_ui_element)`
- `clear_dice_ui()`
- `get_selected_status() -> boolean`
- `change_face_value(idx: int, value)`
- `get_is_rolling() -> boolean`
- `setstartconditions(_pos: Vector3, _rot: Vector3, _time: float)`

3.9.4 Semantics

3.9.4.1 State Variables

- `num_faces`: Traces number of faces on the die.
- `start_position`: tracks starting position of the die on the game board .
- `start_rotation`: tracks starting rotation of the die.
- `start_time`: Tracks time when the dice is created
- `impulse_range`: contains the amount of force applied to the die when rolling.
- `torque_range`: contains amount of torque applied to the die when rolling.
- `velocity_threshold`: sets max speed dice can be moving before it's considers too fast to stop in time.

- `roll_time_limit`: Time limit that the dice is allowed to roll for before a function is triggered.
- `is_selected`: Tracks if the die has been selected by the user.
- `up_threshold`: Threshold for detecting the "upward" ray during raycasting
- `face_rays`: Dictionary that associates the raycast nodes with face values so that the system knows which side of the die is facing up.
- `dice_faces`: tracks each face of the die
- `face_values`: tracks the values corresponding to each face on the die
- `normalCustomTex`: Tracks the physical visual design of die when not selected.
- `selectedCustomTex`: Tracks the physical visual design of die when selected.
- `dice_ui_element`: controls the design of the die's ui icon
- `is_rolling`: checks if the die is rolling
- `roll_start_time`: is the time when the die starts rolling

3.9.4.2 Environment Variables

N/A

3.9.4.3 Assumptions

Assumes that the assets associated with creating the die exist.

3.9.4.4 Access Routine Semantics

- `roll()`
Transition: Physically rolls the die,
- `get_face_value() -> int`
Output: determines which face is facing upwards, and returns that face as the die's value
- `set_dice_ui(ui_element)`
Transition: changes the visible ui design of the die
- `clear_dice_ui()`
Transition: remove the visible changes in the die
- `get_selected_status() -> boolean`
Output Checks if the die has been selected

- `change_face_value(idx: int, value)`
Output: changes the value that the face of the die represents
- `get_is_rolling() ->boolean`
Output: Checks if the dice is still rolling
- `setstartconditions(_pos: Vector3, _rot: Vector3, _time: float)`
Transition: sets up the starting conditions for the die, such as its starting position, it's initial time of creation, etc

3.9.4.5 Local Functions

- `setup_faces()`
Description: Initiates the faces of each die, and sets up ray-cast nodes to each face of the die so that the game can tell which face of the die is facing up.
- `setup_dice_specific_variables(_faces: int)`
Description: initializes the starting parameters and initial state variables for each die of each die
- `_toggle_selection_status()`
Description: detects if the die has been selected by the player or not, and alters the die's visual design accordingly.
- `reset_dice_tex()`
Description: resets the design of the die to it's default, depending on whether the die is selected or not
- `normal_dice_tex()`
Description: sets the physical design of the die to be the default for a non-selected die
- `selected_dice_tex()`
Description: sets the physical design of the die to be the default for a selected die

3.10 MIS of GameSettings Module

3.10.1 Module

GameSettings

3.10.2 Uses

NetworkManager2p

3.10.3 Syntax

3.10.3.1 Exported Constants

N/A

3.10.3.2 Exported Access Programs

- `collectInfo()`

3.10.4 Semantics

3.10.4.1 State Variables

- `hand_settings_saved`: Verifies whether the hand settings for the current game session have been saved.
- `hand_settings_refs`: Dictionary containing the settings for each hand selected by the user during game setup.
- `hand_settings_vals`: Dictionary containing the settings for each hand saved by the user after being selected.
- `game_settings`: Dictionary storing the game settings implemented in the game.
- `dice_count`: Counts the number of dice in the game
- `dice_type`: contains the type of dice used for this game
- `win_cond`: contains the win condition for the game
- `show_opponent_rolls`: Checks if the ability to see opponent rolls is enabled for this game session
- `timed_rounds`: Checks if the option for timed rounds is enables for this game session

3.10.4.2 Environment Variables

N/A

3.10.4.3 Assumptions

Assumes all options are valid

3.10.4.4 Access Routine Semantics

- `collectInfo()`
Transition: Checks if the player is the host for the current game session, and if the player is the host, gives that player the ability to specify the custom game settings for their session.

3.10.4.5 Local Functions

- `_on_start_game_pressed()`
Description: Implements the chosen game settings for the current game session upon the pressing of the start game button.
- `_win_condition_toggled(ID: int)`
Description: Toggles the win condition setting for the game session
- `_on_advanced_settings_pressed()`
Description: When the advanced settings button is pressed, this function opens the advanced settings screen, enabling the user to further customize their game session by adjusting the advanced game settings.
- `_populate_advanced_settings()`
Description: Helper function to help populate the advanced settings screen with advanced game options.
- `save_preset()`
Description: Saves a new preset with its game and hand settings specified by the player
- `load_preset_buttons()`
Description: Loads all existing preset options for selection by the user
- `_on_preset_selected(preset_name: String)`
Description: Loads the game and hand settings of the chosen preset, and applies these settings to the UI.
- `update_ui_fields(game_settings: Dictionary, hand_settings: Dictionary)`
Description: Updates the UI input fields with the loaded game and hand settings.
- `_create_hand_option(hand_name: String, hand_type: Array) -> HBoxContainer`
Description: Helper function that creates the settings and UI option for each hand. It also saves the created settings for each hand within a dictionary of hand settings.
- `save_advanced_settings()`
Description: saves the selected hand settings and closes the advanced settings UI
- `_on_return_to_settings_pressed()`
Description: removes the advanced settings UI and restores the regular game settings UI.
- `_on_roll_visible_toggled(_state: boolean)`
Description: toggles the option that controls whether seeing the opponent's rolls is enabled during the game

- `_on_timed_round_toggled(_state: boolean)`
Description: toggles the option that controls whether the rounds will be timed or not.
- `dice_values_changed(_state: boolean)`
Description: handles the unsaving of any saved hand settings when the dice values are changed, as the previously saved hand settings are no longer valid.
- `copy_hash_to_clipboard()`
Description: Copies the connect code to the system clipboard, allowing a user to paste it somewhere else so that their friends can use that code to connect to the user's game session.
- `_allow_game_start()`
Description: makes the start game button visible, allowing the user to press the button and start the game
- `on_home_pressed()`
Description: returns user to the home screen and disconnects the user from any online game session

7 Appendix

[Extra information if required —SS]

Appendix — Reflection

[Not required for CAS 741 projects —SS]

The information in this section will be used to evaluate the team members on the graduate attribute of Problem Analysis and Design.

The purpose of reflection questions is to give you a chance to assess your own learning and that of your group as a whole, and to find ways to improve in the future. Reflection is an important part of the learning process. Reflection is also an essential component of a successful software development process.

Reflections are most interesting and useful when they're honest, even if the stories they tell are imperfect. You will be marked based on your depth of thought and analysis, and not based on the content of the reflections themselves. Thus, for full marks we encourage you to answer openly and honestly and to avoid simply writing “what you think the evaluator wants to hear.”

Please answer the following questions. Some questions can be answered on the team level, but where appropriate, each team member should write their own response:

1. What went well while writing this deliverable?
2. What pain points did you experience during this deliverable, and how did you resolve them?
3. Which of your design decisions stemmed from speaking to your client(s) or a proxy (e.g. your peers, stakeholders, potential users)? For those that were not, why, and where did they come from?
4. While creating the design doc, what parts of your other documents (e.g. requirements, hazard analysis, etc), if any, needed to be changed, and why?
5. What are the limitations of your solution? Put another way, given unlimited resources, what could you do to make the project better? (LO_ProbSolutions)
6. Give a brief overview of other design solutions you considered. What are the benefits and tradeoffs of those other designs compared with the chosen design? From all the potential options, why did you select the documented design? (LO_Explores)