Lee's Summit North High School

[The Comp. Sci. Crew]

Oct. 13th, 2022

Matthew Allen
Thomas Berhe
Antonio Donato Navato
Julian Somi

01

Problem Statement

According to NASA, a round trip to the moon takes 2 days, 22 hours, and 56 minutes of flight time. Better yet, a one-way ticket to mars demands 9 MONTHS STRAIGHT—or 274 some days—of in-cabin time. With all sincerity, paying space tourists require the wherewithal to stave off the most deadly boredom. Enter the Galactic Arcade. With a selection of multiplayer games that work seamlessly anywhere in the cosmos, every space-faring tourist can expect to open the app and play a fun intellectual game with their fellow cabin mates.

Measurable Benchmarks:

- ➔ The Galactic Arcade offers a variety of game selection.
- ➔ The Galactic Arcade offers multiplayer games.
- ➔ The Galactic Arcade remains playable offline.

With classics such as chess, checkers, and connect four, the Galactic Arcade appeals to all age demographics with an experience to brighten any space journey.

```
22
23  # Initializes the main.board_state data structure.
24  func create_board_state():
25      main.board_state = Dictionary()
26      for row in range(ENDING_COORD_OF_BOARD_LENGTH, STARTING_COORD_OF_BOARD_LENGTH - 1, -1):
27          for column in range(STARTING_COORD_OF_BOARD_LENGTH, ENDING_COORD_OF_BOARD_LENGTH + 1):
28              main.board_state[Vector2(column, row)] = null
29
```

02

Data Structures

The Galactic Arcade contains digital versions of classic hit board games (i.e. chess, checkers, and connect four), necessitating a digital representation of the game board. For explanation purposes, let's examine the standard 64 squared, 8 by 8 chess board, which has many different implementations ranging from hashmaps, 1D arrays, 2D arrays, and even bitboards. Ultimately, our group decided to utilize a hashmap, or dictionary, containing 64 keys, each consisting of a two-element tuple, or Vector2. The dictionary stores references to objects of the piece class or null to denote an unoccupied space. (The construction of the aforementioned data structure is depicted in the screenshot above.)

Before committing to a dictionary, our team heavily considered and partially implemented a 2D array board representation. However, we were persuaded towards dictionaries because the developer interaction, for us, with 2D arrays seemed cumbersome. Rather than indexing a nested array, we found it more human-readable and less verbose to call a dictionary with a Vector2 as the key. Continuing with this trend, we found 1D arrays complicated the movement pattern of pieces, and that bitboards were out of the scope of our current ability.

```
# Utilizes the MOVEMENT constant to return a list of pseudo-legal moves.
func compile_pseudo_moves() -> Array:
    var pseudo_moves = []
    for direction in MOVEMENT:
        var candidate_destination = self.piece_position
        for _cycle in range(16):
            candidate_destination = Vector2(candidate_destination[0] + direction[0],
                                            candidate_destination[1] + direction[1])
            var destination_state = main.access_state(candidate_destination)
            if destination_state is Piece:
                if self.piece_alliance != destination_state.piece_alliance:
                    pseudo_moves.append(main.Move.Capture.new(main, self,
                                        self.piece_position, candidate_destination))
                break
            elif destination_state == null:
                pseudo_moves.append(main.Move.new(main, self, self.piece_position,
                                                  candidate_destination))
            else:
                break
    return pseudo_moves
# ----- ----- ----- ----- -----
```

```

# Calls all the pieces of an alliance to return their pseudo legal moves.
func compile_all_pseudo_moves(alliance):
    var pseudo_moves = []
    for piece in active_pieces:
        if piece.piece_alliance == alliance:
            pseudo_moves.append_array(piece.compile_pseudo_moves())
    return pseudo_moves
# ----- ----- ----- ----- -----
```

Algorithms

The uppermost picture shows the algorithm, *compile_pseudo_moves(),* contained within the Piece class. The *compile_pseudo_moves()* algorithm finds every pseudo legal move available to a piece returning them as an array. In order to accomplish its task, the algorithm iterates through each direction of a piece's *MOVEMENT* constant. On each iteration, it cycles 16 times in the given direction, breaking the loop when the piece falls outside the game board or hits another piece of the same alliance. With each successful *_cycle* the algorithm creates an instance of the move class and appends it to the *pseudo_moves* array.

Algorithms

As a team, we intentionally decided to compartmentalize the compile_pseudo_moves() algorithm within a function because of our general design philosophy. The source code for Galactic Arcade follows an object-oriented approach instead of a procedural one. Additionally, the compile_pseudo_moves() function is called hundreds of times within one play session. If the compile_pseudo_moves() algorithm was not contained in its own function, code duplication would become excessive.

The *compile_all_pseudo_moves()* function within the *main.gd* file calls the *compile_pseudo_moves()* algorithm of all active pieces collecting the pseudo-legal moves of an alliance in the process. With the goal of the compile_pseudo_moves() function being to abstract away the lower level details of generating pseudo-legal moves, the compile_all_pseudo_moves() function confirms that purpose by allowing us the developers to quickly retrieve a list of all the pseudo-legal moves of an alliance. (We apologize for the verbose wording.)

Galactic Arcade Written Requirements

- **Four Pillars of Object Oriented Programming**
  a. Screenshot or copy and paste sections of code that demonstrate each of the four pillars of encapsulation, abstraction, inheritance and polymorphism.

```gdscript
# Blueprint for a king checker piece's behavior.
class King extends Piece:
	const MOVEMENT := [Vector2(-1, -1), Vector2(1, -1), Vector2(-1, 1), Vector2(1, 1)]

	func _init(main = null, piece_gui = null,

	# Utilizes the MOVEMENT constants to return a list of pseudo-legal moves.
	func compile_moves() -> Array:
		var captures = []
		var moves = []
		for direction in MOVEMENT:
			var candidate_destination = self.piece_position
			var candidate_path = Vector2(candidate_destination[0] + direction[0],
					candidate_destination[1] + (direction[1]))
			candidate_destination = Vector2(candidate_destination[0] + (2 * direction[0]),
					candidate_destination[1] + (2 * direction[1]))
			var path_state = main.access_state(candidate_path)
			if (path_state is Piece and path_state.piece_alliance != self.piece_alliance and
					main.access_state(candidate_destination) == null):
				captures.append(main.Move.Capture.new(main, self, path_state,
						self.piece_position, candidate_destination))
			if captures.size() > 0:
				return [captures, moves]
		for direction in MOVEMENT:
			var candidate_destination = self.piece_position
			candidate_destination = Vector2(candidate_destination[0] + direction[0],
					candidate_destination[1] + direction[1])
			var destination_state = main.access_state(candidate_destination)
			if destination_state == null:
				moves.append(main.Move.new(main, self, self.piece_position,
						candidate_destination))
		return [captures, moves]
	# ----- ----- ----- ----- -----
```

```gdscript
# Blueprint for a pawn checker piece's behavior.
class Pawn extends Piece:
	const MOVEMENT := [Vector2(-1, 1), Vector2(1, 1)]
	var HEADING #: int

	func _init(main = null, piece_gui = null,

	# Utilizes the MOVEMENT constants to return a list of pseudo-legal moves.
	func compile_moves() -> Array:
		var captures = []
		var moves = []
		for direction in MOVEMENT:
			var candidate_destination = self.piece_position
			var candidate_path = Vector2(candidate_destination[0] + direction[0],
					candidate_destination[1] + (HEADING * direction[1]))
			candidate_destination = Vector2(candidate_destination[0] + (2 * direction[0]),
					candidate_destination[1] + (2 * HEADING * direction[1]))
			var path_state = main.access_state(candidate_path)
			if (path_state is Piece and path_state.piece_alliance != self.piece_alliance and
					main.access_state(candidate_destination) == null):
				captures.append_array(check_pawn_promotion(
						main.Move.Promotion.Move_Type.CAPTURE_MOVE, path_state,
						candidate_destination))
			if captures.size() > 0:
				return [captures, moves]
		for direction in MOVEMENT:
			var candidate_destination = self.piece_position
			candidate_destination = Vector2(candidate_destination[0] + direction[0],
					candidate_destination[1] + (HEADING * direction[1]))
			var destination_state = main.access_state(candidate_destination)
			if destination_state == null:
				moves.append_array(check_pawn_promotion(
						main.Move.Promotion.Move_Type.BASE_MOVE, null,
						candidate_destination))
		return [captures, moves]

```

```
32
33  var piece_position setget set_position, get_position #: Vector2
34
35  # Encapsulates the piece_position variable, forcing all interactactions through the function.
36  func set_position(position):
37      piece_position = position
38
39  # Encapsulates the piece_position variable, forcing all interactactions through the function.
40  func get_position():
41      return piece_position
42  # ----- ----- ----- ----- -----
43
```

b. Describe the advantages and disadvantages of using these pillars within your code.

Polymorphism: Polymorphism is essential for creating pieces with similar structures from the Piece class but with altered methods. The pawn and king inherit their attributes and methods from the Piece parent class and override the compile_moves() method with their own to establish their individual movement options.

Inheritance: Inheritance makes coding the pieces significantly less time consuming and allows lenient creation of more. The king and pawn are child classes that inherit their movement behavior from the Piece parent class.

Encapsulation: In order to get and set the position of chess pieces, the function set_position() and get_position() are created. These methods represent encapsulation in the program so whenever a piece's location needs to be obtained or changed, it goes through these methods instead of directly being accessed.

Abstraction: Readability and efficiency is gained by practicing abstraction in this program. The compile_moves() method in the king and pawn blueprints contain all the necessary logical components for movement behavior. When used and read elsewhere in the project, the implementations are hidden away and represented by just calling the method name.