The Design and Implementation of a Super Pac-Man Spinoff in C++

Aaron Fainman (1386259) Taliya Weinstein (1386891)
School of Electrical and Information Engineering, University of the Witwatersrand, Johannesburg South Africa

ABSTRACT: This report presents the design and implementation of the arcade game, Super Pacman, based off of a dog theme in C++17 and using the SFML 2.5.1 framework. The maze is modelled as a grid of row-column coordinates with all subsequent entities modelled as positions within this grid. The code is structured to enable seperation between the presentation layer, game logic layer, and data layer. The game logic layer consists of a `Game` class that uses a state pattern design. Inversion of control is used to allow game states to modify the game objects. Interface inheritance of the game state classes also allows the states to modify the game state itself. The actual implementation of each class is briefly described. These classes achieved the required basic functionality. There are also feature enhancements, including power pellets represented as steaks within the chosen theme, decent graphics, and the display of the the dog's (Pac-Man model) remaining lives. Dog catchers represent the ghosts and are able to intelligently chase the dog using a pathfinding algorithm. A comprehensive unit testing suite has been implemented, with specific testing focused on game objects which move and collide. There are many ways the project could be improved. Regarding, game structure, the many `switch` statements could be extracted into their own classes. Parts of the code could also present better orthogonality. Finally, the game maze would be better represented using a graph structure.

1. INTRODUCTION

This report presents the design, implementation and testing analysis of a C++ based Super Pac-Man spinoff game. The game is designed to adhere to object-orientated programming (OOP) design and modelling principles. All graphics and user interface considerations are handles by the Simple and Fast Multimedia Library (SFML) with the correctness of the solution being verified with a testing suite covering the majority of game objects, with extensive testing observed for movement and collisions of characters.

Section 2 presents a brief background to the original Super Pac-Man game while Section 3 discusses the design considerations followed in producing the final game code. Section 4 deals with the conceptual modelling of the domain. Section 5 describes the functionality achieved in the game executable with Section 6 elucidating upon the code structure employed to meet this functionality. A more detailed account of specific class implementation and dynamic collaboration is presented in Section 7 and Section 8 respectively. Section 9 provides a brief summary of the testing suite and Section 10 presents a thorough evaluation of the provided solution covering its functionality, maintainability and orthogonality.

2. BACKGROUND

The Super Pac-Man game originated in 1982 as an arcade maze-chase game whereby the player-controlled entity, Super Pac-Man, could move through a maze. Super Pac-Man had to avoid ghosts which, upon collision, would decrease the number of lives of Super Pac-Man until either he won (by collecting all the fruit) or died (by reaching zero lives). Within the maze, there are entities with which the Super Pac-Man can interact. These include special entities such as stars, super pellets and power pellets enabling various enhancements to the Super Pac-Man play[1]. Additional game mechanics are described in [2].

3. DESIGN CONSIDERATIONS

3.1 Requirements

- Write a keyboard driven Super Pac-Man game operating upon the same game mechanics of the original game.
- Game design must conform to good OOP conventions and structure.
- Provide a comprehensive test suite which verifies the correctness of the game logic. Special focus should be given to the movement and collision of game entities.
- At the very least, basic Super Pac-Man functionality must be achieved by the game spin off. This functionality is listed in [2].

3.2 Prescribed and Imposed Constraints

- The game needs to be coded using modern idiomatic C++17.
- The game needs to make use of the SFML 2.5.1 exclusively. No frameworks built on top of SFML can be utilised.
- The screen window is constrained to a $1600x900$ pixel size when maximised

3.3 Success Criteria

The solution is considered successful if it is able to implement the above requirements in a way that does not violate any of the prescribed or imposed constraints. The game must run smoothly, with character movement abiding to the inherent rules of the maze and a collision between entities should only be recorded if they have an overlap. Good coding principles, including separation of concerns and concise classes, need to be followed.

3.4 *Game Theme*

The game theme implemented in the Super Pac-Man spinoff is that of a dog in a maze environment that is being chased by dog-catchers. For the remainder of the report the dog-catchers will be referred to as catchers for brevity. The dog has lives which are decremented upon being caught by a catcher. To win the game, the dog must eat all the bones within the maze and avoid being caught by catchers. The key translations of terms from Super Pac-Man into the dog-chatcher domain are listed in Table 1. All of the original game mechanics from the original Super Pac-Man game have been conserved.

Table 1: Translation of Pac-Man Entities to the Dog-Catcher Game Theme

Super Pac-Man Entity	**Corresponding Dog-Catcher Entity**
Super Pac-Man	Dog
Ghost	Catcher
Fruit	Bones
Power Pellet	Steak

4. CONCEPTUAL MODEL OF THE DOMAIN

In modelling the domain, it is important to utilise a ubiquitous language for developing the model that is software language independent and thus implementation independent [3]. Since, the game theme is that of a dog being chased by catchers, as detailed in Section 3.4, the corresponding defined terms in Table 1 are utilised in describing the conceptual model of the given problem domain [2]. Owing to there not being a standard diagram notation for modelling the domain, the decision is taken to represent the model utilising the bare minimum UML diagram class structure with basic multiplicity [4].

The domain model utilised in the design of the dog-catcher game is given in Figure 1. The blue domain entities represent actors within the problem domain space, while the rest of the domain entities serve to demonstrate the key interrelations and vocabulary of the problem domain space [4]. To fully develop the conceptual model of the domain, a more detailed account of the model is described in Section 4.1 for certain ambiguous domain entities. The domain entities, user and score, are considered to have a standard acceptable model and subsequently are not elucidated upon further.

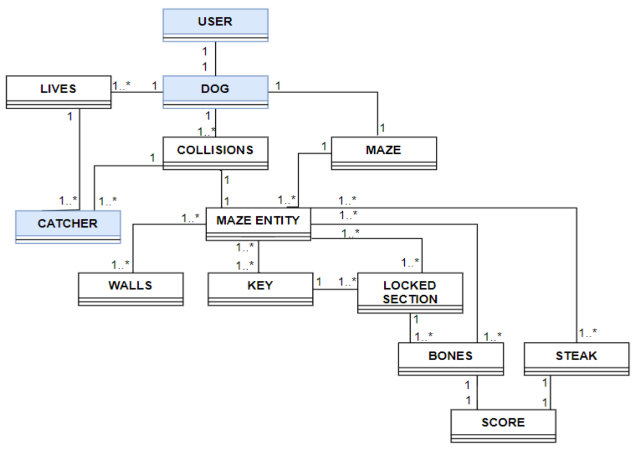

Figure 1: Domain Model Diagram of the Dog-Catcher Game

4.1 *Modelling of Unintuitive Domain Entities*

4.1.1 Maze and Maze Entity: The maze is modelled as that grid consisting of positions in rows and columns. Each position in the grid is indexed by specifying both a row value and a column value. At certain positions within the maze, there are maze entities. These entities are able to be interacted with by both the dog and catcher characters via collisions. The collisions between the dog and maze entities can influence the maze representation. This is the case where the dog collides with a key and unlocks the locked sections. Additionally, in the case of a collision between the dog and either the steak or bone, these maze entities are no longer represented as they have effectively 'been eaten' by the dog.

4.1.2 Catcher: Catcher characters are modelled as malicious, moving positions within the maze. Their positions update based on the current dog character position. This enables the catchers to intelligently follow the dog position through the maze.

4.1.3 Dog and Lives: The dog character is a user-controlled maze explorer modelled as a position within the maze that can be removed through collisions with the catcher character. The number of times the position of the dog can be regenerated within the maze is modelled as a life. Through user input, the position of the dog changes, thus enabling the dog to move through the maze. The dog movement abides to the inherent maze structure. This models physical property interactions of the real world (that is the character cannot pass through a wall or locked section without a key.)

4.1.4 Collisions: The collisions are modelled as all the positions of maze entities (excluding walls and locked sections) that overlap with the position of the dog character. Intercepts between the dog and catcher position decrease the number of times the character can be regenerated within the maze, with the user losing the game when this number reaches zero. Collisions between the dog character and maze entities either alter the maze representation, increases the user score or enable the user to win the game.

5. FUNCTIONALITY

The dog-catcher game has been implemented such that it meets all basic functionality requirements [2]. In addition to this basic functionality the following minor features have been implemented.

- The maze is able to be read in from a file
- The player has more than one life, with the life displayed on the screen
- Graphics are not composed of simple shapes
- Scoring system with display for current game
- The maze contains power pellets with functionality as detailed in [2].

Lastly, the major feature of intelligent ghost movement has been implemented with a detailed description of this functionality given in Section 7.4.3.

6. CODE STRUCTURE

6.1 *Overview*

The game is structured to follow an OOP paradigm. As far as possible, principles of OOP have been followed. In terms of code structure, these included the separation of layers, encapsulation, and polymorphism.

There are three distinct layers in the project: a presentation layer, a game logic layer, and a data layer. The layers have been implemented such that each could be changed without affecting another layer. A summary of this is shown in Figure 2.

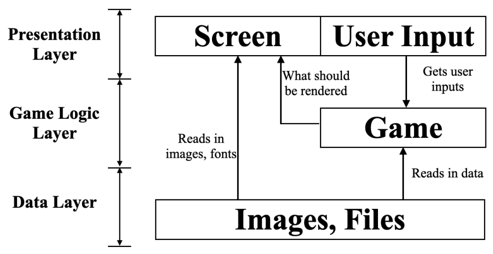

Figure 2: Separation of layers in the game

6.2 *Application Layer*

The presentation layer consists of two classes: one for dealing with the user input and another for rendering all images and text to the screen. This layer makes extensive use of SFML. However, this is the only layer that depends on the library. SFML has many built-in features convenient for dealing with game logic behaviour, such as collision detection and clocks. This being said, the game logic layer has been decoupled from the library so that if the graphics library is changed, only the presentation layer need be updated.

The application layer follows a state pattern design. This design pattern models the different states the game may take on while running. Each state corresponds to its own class, with the game's state being updated when certain conditions are fulfilled. The advantage of this is that long conditional statements need not be used to consider every possibility in every state. Additionally, each state can update what the next state ought to be and new states can be added without needing to change the actual game. Maintainability of code is thus also improved.

With this in mind, the game logic layer consists of three different broad categories. The first category is the set of classes responsible for dealing with different game states. A single abstract base class, `GameState`, exists. Derived from this are classes that take care of each unique state of the game. This has allowed for polymorphism where the main game class runs the same loop regardless of the game state. Those states that are dealt with include when the game is starting up (`GameStart`), running (`GameActive`), over (`GameOver`), paused (`GamePaused`), and when the dog has eaten a steak (`GameSteak`). The second category in the game logic layer are the actual game objects, such as a `Dog`, `Catcher`, and `Maze`. Finally, the last category consists of helper classes, responsible for taking care of calculations and behaviours common to different game objects. Some examples include `Position` (for dealing with x-y coordinates of the `Screen`), `GridPosition` (for dealing with row-column coordinates of the `Maze`), and `Movement` (for dealing with how the characters traverse the maze).

6.3 *Data Layer*

The data layer consists of the files, images, and fonts required for running the game. There are a handful of classes responsible for reading this data. In addition, there is a header file, `CommonConstants`, that contains a set of game constants. These must be accessible to all classes in the game. There are enumerations, such as `direction`, which is used extensively by any object that deals with movement, and collisions. There are also constants that should not be varied. A property like the character size, should be universal throughout the project. Additionally, the game is structured and should be understood as being a grid through which characters move and over which textures are drawn. The size of this grid must also be known. The advantage of using a common file is that any code that relies on some of these game properties need not be rewritten if that property is changed. Only the header file constant needs to be changed. Universal constants should be carefully chosen, however, so that the internal details of classes remains hidden. For example, the file directory for an image, should only be known by the class in the presentation layer that requires it. It is an internal detail that is not necessary for other classes. This is discussed further in the implementation of the `Screen` class in Section 7.2.2.

7. CLASS IMPLEMENTATION

This Section is primarily focused on the individual class implementation used in the dog-catcher game. As such only very basic dynamic interactions are discussed where necessary in describing the classes' responsibilities. A more detailed account of the dynamic collaborations occurring in the game can be found in Section 8..

7.1 *Game and Game States*

7.1.1 GameState: `GameState` defines an abstract base class for all possible game states. This base class serves as a contract that all derived GameStates classes must satisfy in order to run the game. Figure 4 demonstrates this contract. Each derived class must handle user inputs (the `handleInput` method), update the game characters (`updateCharacters`), handle collisions in the game (`handleCollisions`), and update the screen object with what drawable objects should be rendered (`draw`). All the methods are a void return type. Two of the methods, `handleInput` and `handleCollisions`, take in a `GameState` argument as a reference parameter and can modify the game's state if needs be. There is no sequential order with which these methods should be called when running the game.

7.1.2 GameStart: This class, derived from `GameState`, takes care of the game when starting up. It resets all game parameters (number of lives, score, and maze) to their defaults, and tells the `Screen` object to render the splash screen. The only input it handles is if the user presses Enter to start the game.

7.1.3 GameActive: `GameActive` is responsible for the normal running of the game. It tells the `Screen` object to render the maze, all characters, the score, and the number of lives. `GameActive` handles the user pressing the Space bar to pause the game. This updates the game state to be `GamePaused`. It also handles arrow keys being pressed to change the dog's direction of movement. This class allows characters to update by calling every character's update function. Collisions between all game objects are then managed based off of type of collision returned by an object of the `Collisions` class. Section 8. further discusses these collisions and the flow of states.

7.1.4 GameSteak: This class runs the game when the dog eats a steak. The state only lasts for a finite period of time (taken as 5 seconds), after which the game is returned to being in `GameActive` state. `GameActive` and `GameSteak` have very similar roles. In fact the key difference between the two is that when a dog and catcher collide in `GameSteak`, the catchers respawn and points are awarded. Arguably, `GameSteak`, need not be its own class but rather a subcase in `GameActive`. The decision was made to keep `GameSteak` as its own class for maintainability reasons. If in the future, the rules around collisions or rendering were to be changed after a dog ate a steak, then a separate class would be more useful. This consideration is further analysed in Section 10.

7.1.5 GamePaused: When the game is paused, from the user pressing the Space bar, the state becomes `GamePaused`. This state will not update the characters or handle collisions. It only tell the `Screen` object to renders the pause screen and returns the game back to being active after the user unpauses (by again pressing the Space bar).

7.1.6 GameOver: Similar to `GamePaused`, the `GameOver` state neither updates the characters nor deals with any collisions (the function definitions are left empty). Its only role is to have the game over drawn to the screen and restart the game when the user presses Enter.

7.1.7 Game: `Game` is a small class that handles setting up the game. It instantiates all game objects, and then runs a game loop where it calls the methods of `GameState` described above. It does not update its state, however. That is left to the game state classes. Another responsibility is getting the user input from an object of the `UserInput` class and handling if the user closes the window. All other user inputs are communicated to the game state object. Section 8. further discusses the dynamic interaction between `Game` and the `GameState` objects.

7.2 *Presentation layer*

7.2.1 UserInput: `UserInput` is a class which models the key-press events of the user in moving the dog character. The chief responsibility of `UserInput` is capturing the key pressed by the user from the allowed keys defined. These keys include the arrow keys, the Enter key and the Space bar. All other keys pressed are subsequently treated as if no key has been pressed. `UserInput` is also able to capture when a user closes the game window. These user interactions with the game are captured in a `scoped enum` which is subsequently used for display purposes in `Screen` and for informing `Game` that the user has exited the game window.

7.2.2 Screen: The `Screen` class is utilised for purely display purposes to demonstrate the interactions between game entities on an SFML generated window. Subsequently, this class acts as a feedback class demonstrating the effect of the user input on the game environment to subsequently inform future user inputs. The `Screen` class contains all the constant paths for loading `sf::Textures` as constant private members to aid in information hiding. Its responsibilities include polling the `sf::Events` for user in-

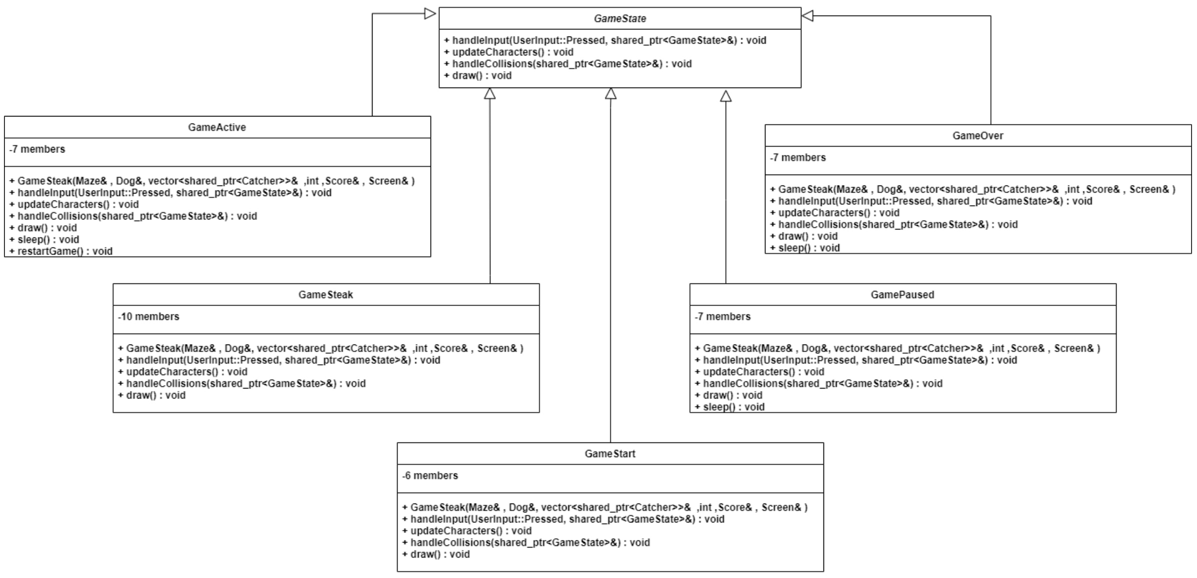
GameState
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void
GameActive
-7 members
+ GameSteak(Maze& , Dog&, vector<shared_ptr<Catcher>>& ,int ,Score& , Screen&)
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void
+ sleep() : void
+ restartGame() : void
GameOver
-7 members
+ GameSteak(Maze& , Dog&, vector<shared_ptr<Catcher>>& ,int ,Score& , Screen&)
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void
+ sleep() : void
GameSteak
-10 members
+ GameSteak(Maze& , Dog&, vector<shared_ptr<Catcher>>& ,int ,Score& , Screen&)
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void
GamePaused
-7 members
+ GameSteak(Maze& , Dog&, vector<shared_ptr<Catcher>>& ,int ,Score& , Screen&)
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void
+ sleep() : void
GameStart
-6 members
+ GameSteak(Maze& , Dog&, vector<shared_ptr<Catcher>>& ,int ,Score& , Screen&)
+ handleInput(UserInput::Pressed, shared_ptr<GameState>&) : void
+ updateCharacters() : void
+ handleCollisions(shared_ptr<GameState>&) : void
+ draw() : void

Figure 4: Inheritance Hierarchy of GameStates

put, updating the representation of the game accordingly and subsequently displaying this representation. The `Screen` class is implemented as a friend class of `UserInput` due to its needing to access the constantly changing private `sf::RenderWindow` member variable of `UserInput`. Although, these two classes could have been combined into one class, the decision to conserve separation of concerns takes precedence resulting in the current structure described. A critical evaluation of this design decision is presented in Section 10.

7.2.3 Drawable: The abstract `Drawable` base class is used to model all entities within the game which are required to be represented on the screen. These specific Drawable entities are discussed in Section 7.3. The `Drawable` class has one pure virtual function to retrieve a drawable entity's position as well as its representation as seen in Figure 5.

7.3 Drawable Entities in the Game

Figure 5 demonstrates the inheritance hierarchy of the drawable entities, clearly indicating that there are two main subtyppes which inherit from the abstract `Drawable` base class: `Maze` and `Character`. These subtypes are discussed in more detail below.

7.3.1 Character: The `Character` class is an abstract base class which is used to model all drawable entities within the maze which are capable of moving. These entities are the `Dog` class and the `Catcher` class. The `Character` class has pure virtual functions responsible for retrieving the position of the moving entities (`getPosition`), unpausing the moving entities once game play has began (`unpauseCharacter`) and moving the moving entities back to their starting location when restart triggering events occur (`respawn`).

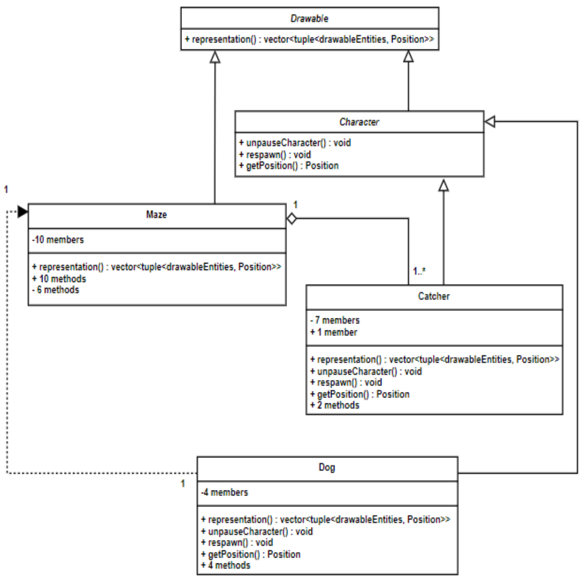
Drawable
+ representation() : vector<tuple<drawableEntities, Position>>
Character
+ unpauseCharacter() : void
+ respawn() : void
+ getPosition() : Position
1
Maze
-10 members
+ representation() : vector<tuple<drawableEntities, Position>>
+ 10 methods
- 6 methods
1
1..*
Catcher
- 7 members
+ 1 member
+ representation() : vector<tuple<drawableEntities, Position>>
+ unpauseCharacter() : void
+ respawn() : void
+ getPosition() : Position
+ 2 methods
1
Dog
-4 members
+ representation() : vector<tuple<drawableEntities, Position>>
+ unpauseCharacter() : void
+ respawn() : void
+ getPosition() : Position
+ 4 methods

Figure 5: Drawable Entity Class Interactions

7.3.2 Dog: `Dog` is a class which is used to model the user's desired movement within the maze which conforms to the rules of the maze discussed in Section 7.4.2. As a subtype of both `Drawable` and `Character`, `Dog` overrides the representation retrieval function as well as the above mentioned functions for `Character`. In addition to these overwritten functions, `Dog` also has the responsibility of updating its position based on the allowed movement. This allowed movement required knowledge of the maze environment, and subsequently there is a weak dependency between `Maze` and `Dog`. The `Dog` has a default movement direction of right upon game start up.

7.3.3 Catcher: The Catcher class is also a subtype of both Drawable and Character and subsequently overrides both these classes' virtual functions. The key differentiating factor between the Dog and the Catcher classes is the manner in which both character's move. While the Dog movement is governed by the user and the inherent structure of the maze, the Catcher's default movement is to follow the Dog through the maze in an attempt to collide. This movement requires knowledge of the maze and subsequently results in an aggregation relationship between the Catcher and the Maze. The intelligent catcher movement is discussed in Section 7.4.3. The Catcher can also be constructed such that it moves in a random direction through the use a random number generator to pick an allowed direction with the Movement class.

7.3.4 Maze: The Maze utilises a scoped enum of mazeEntities to model the static maze environment and provides the core feedback necessary for informing the characters of allowable movements and collisions. As such it has a large range of responsibilities which include retrieving the starting location for all characters and throwing an exception if more than one dog starting positions are found. Additionally, the Maze is responsible for informing Game when there are no more bones remaining, signifying that the user has won, and with what specific maze entity the Dog collided. As with other Drawable entities, Maze is responsible for its representation and updating. It is also able to reset itself to the starting version when a new game is requested. Lastly, Maze has the functionality to convert between different coordinate spaces enabling the conversion from its native rows and columns to the x-y coordinate space of the Screen and the character corner locations also as x-y coordinates.

7.4 ***Helper Classes***

7.4.1 Position and GridPosition: These two classes are used for handling calculations on positions. Examples include adding two positions together, adding a distance in a direction, and dividing a position by a number. Position and GridPosition have similar roles, however, Position works in x-y coordinates and GridPosition works in row-column coordinates.

7.4.2 Movement: Movement is a helper class that deals with updating a character's positions with the structure of the maze. The game terrain (ie. the maze) is defined in Maze. Movement just coordinates the movement within this terrain by checking a character's new position with that region in Maze. The class stores the direction being travelled, the character's speed, and the character's position. The public member function, updatePosition, takes care of finding the distance to be moved. It uses an internal clock to check the time between updates so that a character's speed remains constant regardless of the computer that runs the game. This function also checks whether the new position of the character is invalid (such as if the character hits a wall), and returns the new position. Movement also deals with requesting a change in direction. If, for example, a character is moving left and is surrounded by walls above and below, then the character's direction should not be able to be changed to move upwards or downwards. A change in direction is valid if the two corners of the character, in the new direction, can move to the next tile. An example of this is shown in Figure 6.

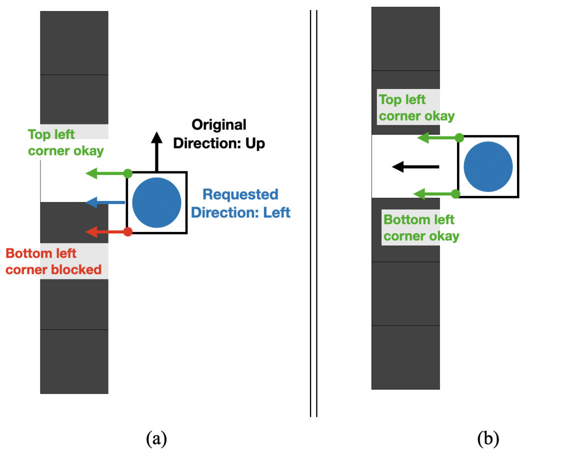

Figure 6: Diagram illustrating how Movement validates a change in direction. A character that would like to move left, should have its two left corners (top and bottom left) able to travel left. If its movement is blocked (as in Figure (a)) then the request is stored until the character is able to move left (as in Figure (b))

There are a few other methods within Movement. These include returning all the possible directions in which movement can occur based off of the given position, and restarting movement after the game has been paused. Additionally, Movement automatically handles wrapping around the screen so that if a character moves off of one edge of the maze, they reappear on the other edge.

7.4.3 Catcher Movement: Catcher movement includes functions that take care of the two possible ghost movement strategies: random movement, and intelligent movement.

Random movement takes in the possible directions in which a character could move. A new direction is produced based on a pseudorandom number generated. There is some nuance in this. If the character can only move back and forth (two opposite directions) then its current direction of movement will not change. Otherwise the character will just oscillate back and forth. Similarly, if the catcher can move in four directions its movement should not change to prevent this oscillation.

The intelligent movement works off of a pathfinding tool. It takes in two positions: the current and required position, alongside the possible directions the character could move in. It returns the best direction based off of the minimum distance to the required position. Ideally pathfinding algorithms are implemented on graph data structures. However, nowhere in the project is the game terrain stored as a graph, and thus pathfinding is less than ideal. A search, adapted from breadth-first searches, has been implemented [5]. This algorithm is based off of a 2D vector of 1s and 0s, representing where a character can travel. The search considers the starting point in the grid and required end point. It moves along the grid, adding every adjacent grid point to the current grid point before considering the next grid point it has stored. If the grid point is blocked then it is removed from the queue. This continues until the required point is found. It has thus stored every grid position between the starting point and end point, and terminated at the first instance the end point is reached. Thus, the shortest path exists somewhere in the collection of points stored. An example of this is shown in Figure 7.

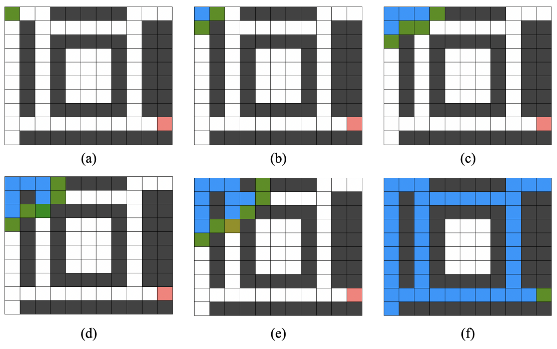

Figure 7: Grid search until the desired point is found. Figure (a) shows the starting point in green and end point in red. Tiles that are blocked are shaded in grey. Figures (b) - (e) demonstrate the path finding. The current set of tiles being examined is shown in blue, and all adjacent tiles (green) are added to the queue. If the tile is blocked it is removed, and the next tile in the queue considered. This continues until the end tile is found (f).

The actual path can then be found. Starting from the end point, the next point in the queue is checked to determine if it is adjacent to the point being considered. If it is then it is kept and becomes the point being considered. If it is not adjacent, it is removed. Adjacency is based off of 4-connectivity (mirroring the game's movement). This continues until the start point is found - an event which signifies that a path has been found. The direction for the character to move in is chosen by considering the length of the path to the end point if the character were to move in that direction. An example of this is shown in Figure 8. If no direction is found, the best direction is chosen (for example, if the desired point is roughly leftwards then the best direction to move in is left). Finally, if no best direction is found then the character should move randomly in any possible direction that conforms to the rules of the maze.

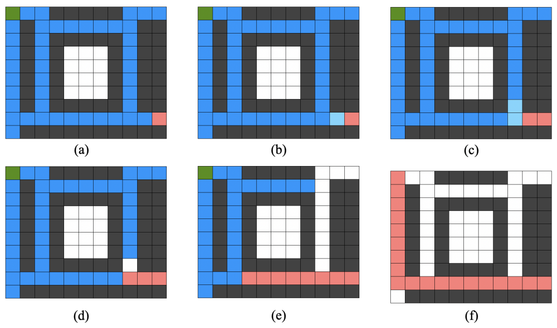

Figure 8: Path derived from the grid search. Figure (a) shows all the sets of points in the search in blue. The current point of interest is shown in red (end point in the path), while the desired point (start) is shown in green. Figures (b) - (e) demonstrate the removal of non-adjacent points. Points in blue are considered and if not directly adjacent (4-connectivity) then it is removed. If it is adjacent, it becomes the new point being considered. This continues until (f) when the start point has been found and the path determined.

7.4.4 FileReader: This class is able to read in a file containing numbers, and return a 2D vector containing those numbers. An exception is thrown if the file cannot be opened.

7.4.5 Collisions: The `Collisions` class models an overlap between the current position of the `Dog` and another entity from `Drawable`. The main responsibilities of `Collisions` is to determine what collision has occurred and subsequently to be able to report this collision data. The determining of the specific type of maze entity collision is performed in `Maze` while `Collisions` is responsible for checking when a collision occurs with a catcher in addition to with which catcher the collision occurred (this is necessary for the collision between the dog and steak where the dog is enabled to reset the catcher's position).

In determining whether a collision has occurred between the catcher and the dog, the positions given as black dots in Figure 9 are checked. If the absolute difference between the catcher's position and the dog's position is within a fractional value of the tile size, then a collision is recorded. A collision is demonstrated in Figure 9 between the white and orange boxes due to their coordinates being within the set tolerance while a collision would not be recorded between the blue box despite the sharing of y2 coordinates.

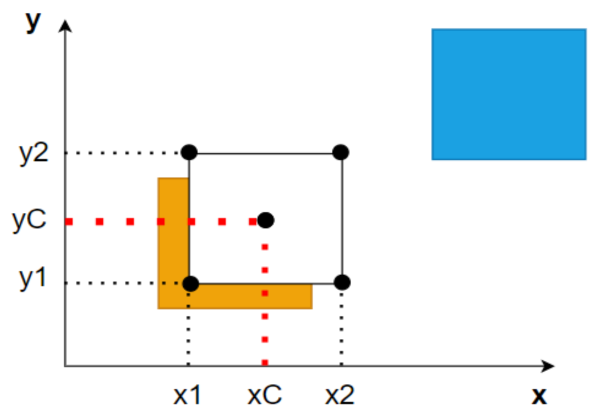

Figure 9: Collision Checking Coordinates with Tolerance. The xC and yC coordinates represent the central position of the white box. Black circles represent corner positions of characters. All black dot positions are compared to determine whether a collision has occured.

7.4.6 Score: This class helps keep track of the score. It stores and updates the current score. A file of high scores can also be read and client code told whether a high score has been reached.

8. DYNAMIC COLLABORATION

The game runs through the dynamic collaboration between all the game objects. For this to happen, the game objects need to be able to communicate with each other. The guiding principles in facilitating these object conversations has been: "ask, do not tell", and "composition over inheritance". The former principle maintains that an object must tell another object what to do, as opposed to calling multiple functions and mutator methods to try have the other object do what is required. The latter principle has been widely implemented in this project. Nearly every class stores other objects as private member variables. For exmaple, in the case of `Catcher` and `Dog`, both have their own unique `Movement` object. On the other hand, many objects store as a constant reference a `Maze` object parameter. This means they have access to the game maze as it is updated, but cannot modify it in any way (although it is up to maze to restrict its public interface so that it cannot be modified unnecessarily).

Composition also features in the game state inheritance structure. New `GameState`-derived objects are constructed by passing in the game objects (`Dog`, `Catcher`, `Score`, and so on) in the constructor. The game state objects can then modify and update the game objects as necessary. This is known as inversion of control [6]. The alternative would be for `Game` to modify its member objects and attempt to deal with every possible scenario. Rather, it passes the flow of control to other objects (the game state objects) using a dependency injection [6]. The flow of these game states is discussed below.

8.1 *Collisions and Game States*

There are two ways that the game's state can be updated. The first is directly by the user input. If the user chooses to end the game (by closing the window) then the `Game` will break the `run()` loop and the window will close. If the game is in startup and the user presses the Enter key, the game state will become active. Finally, if the game is running and the user chooses to pause the game then the game state will be changed to be of type `GamePaused` until the the user chooses to unpause the game.

The other way that the game's state can be updated is through collisions while playing the game. These collisions are:

- Dog-Bone: If the dog eats a bone then the bone is removed from the maze and the score is updated. If there are no bones left then the game is won and the game state is updated to be `GameOver`.
- Dog-Key: If the dog eats a key then the key is removed from the maze, and in turn the maze unlocks a door.
- Dog-Steak: If the dog eats a steak then the game state becomes `GameSteak`, which changes the colour of the catchers and enables the dog to 'eat' them.
- Dog-Catcher: If the dog collides with a catcher while in `GameActive` then the game momentarily pauses, a life is lost, and all characters respawn. If there are 0 lives left then the game state is updated to be `GameOver`. If the game is in `GameSteak` state then the dog-catcher collision will not cause the game state to change and only the catcher will respawn.

Except for the Dog-Key collision, any collision may change the game state. Each game state deals with the collisions as it sees fit.

Figure A in the Appendix shows the sequence diagram when the game is active. Only the active game state is represented since this is the game state with the most dynamic interactions between game objects. The sequence diagram in the steak state is very similar to that of game active. The key difference is just in the handling of collisions. The sequence diagrams for starting the game, ending the game, and pausing the game are trivial. These three states only process user input to change the game state and render images to the screen and are subsequently not included in the sequence diagram.

9. UNIT TESTING

Table 2: Unit Test Summary

Object	Tests	Assertions
Drawable Entities		
Catcher	6	8
Dog	5	6
Maze	23	31
Movement and Collisions		
Collisions	16	26
Movement	12	20
CatcherMovement	7	7
Helper Classes		
FileReader	2	3
Position	8	8
GridPosition	8	8
Scores	4	5
Game States		
GameActive	5	8
GameSteak	5	8
TOTAL	**101**	**138**

Unit testing using the doctest framework is performed on the majority of objects to verify that correct functionality is achieved. Most public functionality of these tested objects is performed directly, yet in some cases indirect testing is required whereby the results of a public function are then utilised. Table 2 summarises the number of tests and assertions performed on all tested objects. In the cases where object interactions are tested, these tests are recorded in the object with the main testing functionality (for example the `Maze-FileReader` is recorded as a `Maze` test as the `Maze` contains the main function used for testing this interaction.) The `Movement`,`CatcherMovement` and `Collisions` classes were thoroughly tested, with these test combined accounting for 35 test cases and 53 assertions .

Not all the functionality is tested. Classes which are not tested include Screen and UserInput, with the Game class only being tested indirectly through the the game state classes `GameActive` and `GameSteak`. The other game states (GameStart,`GamePaused` and `GameOver`) are not able to be tested. These untested classes required simulating user-keyboard interactions that are directly coupled to the SFML framework. Although, there are ways to test these interactions, such as through the implementation of a mocking framework, project time constraints did not enable these testing methodologies to be considered.

10. CRITICAL EVALUATION

10.1 *Functionality*

The game is successful in implementing the basic functionality required [2] as well as the discussed minor and major features in Section 10.1 This success declaration is based on the fact that the game executable runs without a disruption, the characters move smoothly, collisions occur only when two drawable entities overlap and collisions trigger the correct sequence of events. Additionally, there is a splash-screen which is presented in line with the theme which includes the playing instructions.

The functional limitations in the game executable stem from not having the additional major features of a super-pellet state and the ability to edit the maze within the game executable [2]. Although the super pellet state is a natural extension from that of the functionality achieved, time constraints prevented this major feature's execution. The maze editing would not be able to be achieved easily from the current version of the code.

10.2 *Unit Testing*

As a matter of principle, unit testing should occur quickly due to the number of times these tests will be run as well as the potential for more unit tests to be added in future. More tests result in a longer running duration, which when coupled with an already lengthy running could make future development more difficult [7]. In the project's unit testing, the testing suite takes at least 17s to complete. This lengthy execution is due to the movement tests requiring sleep wait times to be implemented to allow the character's to update their position. Additionally, in testing the game states, `GameActive` and `GameSteak`, a screen is constructed which is hen subsequently closed in the test. These state tests thus take a substantial amount of time.

10.3 *Maintainability of Design*

In assessing the maintainability of the code, it is important to consider how easy it would be to change the existing code structure for new feature inclusion as well as the degree of ease with which the code can be debugged. As mentioned in Section 10.1, the maintainability of the code base from a functional perspective is highly dependent on the functionality to be included. Any additional game state would be easily achievable due to the design decision to have all game states fulfil the contract specified in the abstract base class `GameState`.

The code base follows "Don't Repeat Yourself" (DRY) principle, where possible, which enables the code base to be changed in one location exclusively. This is especially relevant in the case of the common constants header file. There are, however, instances within the implementation of the virtual functions of the `GameState` base class whereby implementation is repeated. Although, this could potentially have been eliminated through the utilisation of mix in classes or a composition relationship, it is reasoned that the benefit of having the interface relationship between `GameState` and its derived classes enables the addition of alternative game states, as mentioned above, thus rendering the compromise of repeated code acceptable. This is once more the case in the repetition of code in the implementation of `Character`.

Although, the aim in the design process is to have well-modelled abstractions making use of small classes with distinct responsibilities, this has not always been executed. Examples of this failing in the current project code include the extensive use of `scoped enums` and switch statements in the design, especially when modelling the maze entities. This would subsequently make the addition of other maze entities harder and runs the risk of creating long functions due to the switch statement implementation. This approach is further elucidated upon in Section 10.5.

From a function perspective, all long functions have been split up to ensure that each function fulfils its base responsibility as specified in the name. Short functions are a key feature of code that is able to be easily debugged. This is an advantageous maintainability feature in the code base implemented.

10.4 ***Orthogonality and Coupling of Design***

The design achieves a good separation of concerns with distinctive classes responsible for each responsibility layer. Care has been taken to decouple the game logic implementation from that of the SFML library such that only the presentation layer, consisting of `Screen` and `UserInput`, utilises this library. This decoupling from SFML enables a high degree of orthogonality as only a limited amount of refactoring of the aforementioned classes would be required if a different graphics library were to be utilised in future.

Although the friend class implementation of `Screen` and `UserInput` eliminates the need for accessor and mutator functions, it creates tight couping between these two classes. This tight coupling subsequently reduces encapsulation and ultimately would result in excessive changes to one of the classes if the other class changed. It also shows poor information hiding. However, this being said, `UserInput` and `Screen` are tightly coupled as a result of SFML's implementation of events. Events are polled from the actual SFML window. If either `Screen` or `UserInput` were changed, the latter would need to be changed regardless of whether better encapsulation was achieved.

There are other aspects of the game that show tight coupling. A prime example of this is `Collisions` and `Movement` with `Maze`. The former two classes rely on the fact that the maze is stored as a 2D grid. If, for example, maze's internal representation were to changed to be a graph structure, or work in x-y coordinates as opposed to row-column coordinates, then large parts of `Collision` and `Movement` would need to be reworked. Tight coupling like this shows poor object-oriented design. This being said, parts of the game show good orthogonality. If, as in the example above, maze's internal representation of the game terrain were changed then none of the data or presentation layer would need to be changed. Additionally, `Game`, `Dog`, `Catcher` and all `GameState` objects would remain unchanged.

Finally, there are the `Position` and `GridPosition` classes. Both classes consist largely of mutator methods ("setters") and accessor methods ("getters"), which is generally considered bad OOP. Other objects have access to the internal details of the class. Thus, they may end up coupled to the classes' internal representations. This is known as inappropriate intimacy, or more affectionately, "object orgy" [8]. However, the classes are simple, consisting of a single private member variable that stores the position as a tuple of integers. Thus, issues with inappropriate intimacy, such as hard maintainability or unknown states, are unlikely to occur. Use of these classes is also in line with the DRY principle, in that calculations that would need to be consistently repeated when working in position space can be handled once.

10.5 ***Improvements***

There are a few aspects of the project that could be improved. In terms of the code structure, a better object-oriented design could be implemented. While polymorphism and composition feature throughout the project there are many places where coupling is too tight. Additionally, the code features many `switch` statements, such as in the type of collision (dog-key, dog-bone, etc.) that has occurred, or the type of maze entity (bone, lock, key, etc.). Going forward, these sub-cases should be extracted into their own classes and the operations performed either on or in the objects themselves. This was considered during the later stages of development. However, the technical debt has already been incurred. There was too little time to refactor these sub-cases into their own code.

Another place the design could be improved is in maze's internal representation of the game terrain. There are quite a number of functions that work out where a character could move based on its current position. A far more natural way of representing the maze would have been using a graph structure. Each point on the grid could be represented by a node, and the path between points on the grid represented by edges connecting the nodes. Another benefit of this would have been in the use of `CatcherMovement`'s pathfinding function. There are many pathfinding algorithms (such as A^*) that work on graphs and are more efficient than the one implemented (although for a simple game like this, the implemented pathfinding function works well enough).

11. CONCLUSION

From a functionality perspective, the basic requirements for the game have been met. Certain minor and major feature enhancements are also present. The game consists of bones in sections that are initially locked. These can be unlocked with a key. Catchers exist in the game and intelligently chase the dog. Collisions between the two result in a life being lost and the characters respawning. There are also steaks scattered throughout the maze that make the dog immune to the catcher for a finite period of time. A comprehensive unit testing suite has been implemented. This has tested every class that does not require special testing methods of the graphic or SFML libraries. The game's functionality could be expanded on. The implementation of a super pellet is a natural extension of the current functionality. Additionally, a graph structure should represent the maze. This would improve the efficiency of the intelligent catcher movement.

In terms of game structure, the game uses a state pattern design. Each game state is represented by its own class. A dependency injection of the game objects into these different game states is performed in the game state constructor. This allows the states to run and modify the game as necessary, in a software paradigm known as inversion of control. This flow of control is a better alternative to having a single game class attempt to handle every game state and every case that may arise. Additionally, it improves maintainability in that new game states are easier to implement. The game structure also presents good separation of layers. Game logic and presentation layers are uncoupled. If the SFML graphics library were to be changed, nothing in the game logic layer would need to change. Additionally, all files, fonts, and images are stored in a data access layer. Game constants are stored in a header file that can easily be changed without changing any classes. This being said, there are ways the game structure could be improved. The dynamic collaboration between game objects requires a number of `switch` statements. These should rather be classes in and of themselves.

REFERENCES

[1] 'Super Pac-Man: Classic Arcade Game Video, History Game Play Overview', Arcade Classics. https://www.arcadeclassics.net/80s-game-videos/super-pac-man (accessed Oct. 08, 2020).

[2] S. Levitt, 'Project 2020 - Super Pac-man'. University of the Witwatersrand, Johannesburg, 2020.

[3] E. Evans, *Domain Driven Design: Tackling Complexity at the Heart of Software*. Westford, Massachusetts.: Adison Wesley, 2004.

[4] BigLeap Solutions, 'Domain Model - Part A', Youtube, May 27, 2016. https://www.youtube.com/watch?v=M1e2XwSADDElist (accessed Oct. 06, 2020).

[5]

[6] Reader Man Blog, Inversion of control (IoC), Dependency injection (DI), 2011, https://readerman1.wordpress.com/2011/07/10/ioc_di/ (accessed Oct. 06 2020).

[7] T. Ottinger and J. Langr, 'F.I.R.S.T', Agile in a Flash, Feb. 2009. https://agileinaflash.blogspot.com/2009/02/first.html (accessed Oct. 07, 2020).

[8] John Hopkins Programming Languages Laboratory, 'Refactoring'. . https://pl.cs.jhu.edu/oose/lectures/refactoring.shtml (accessed Oct. 6 2020).

Appendix

A Sequence Diagram

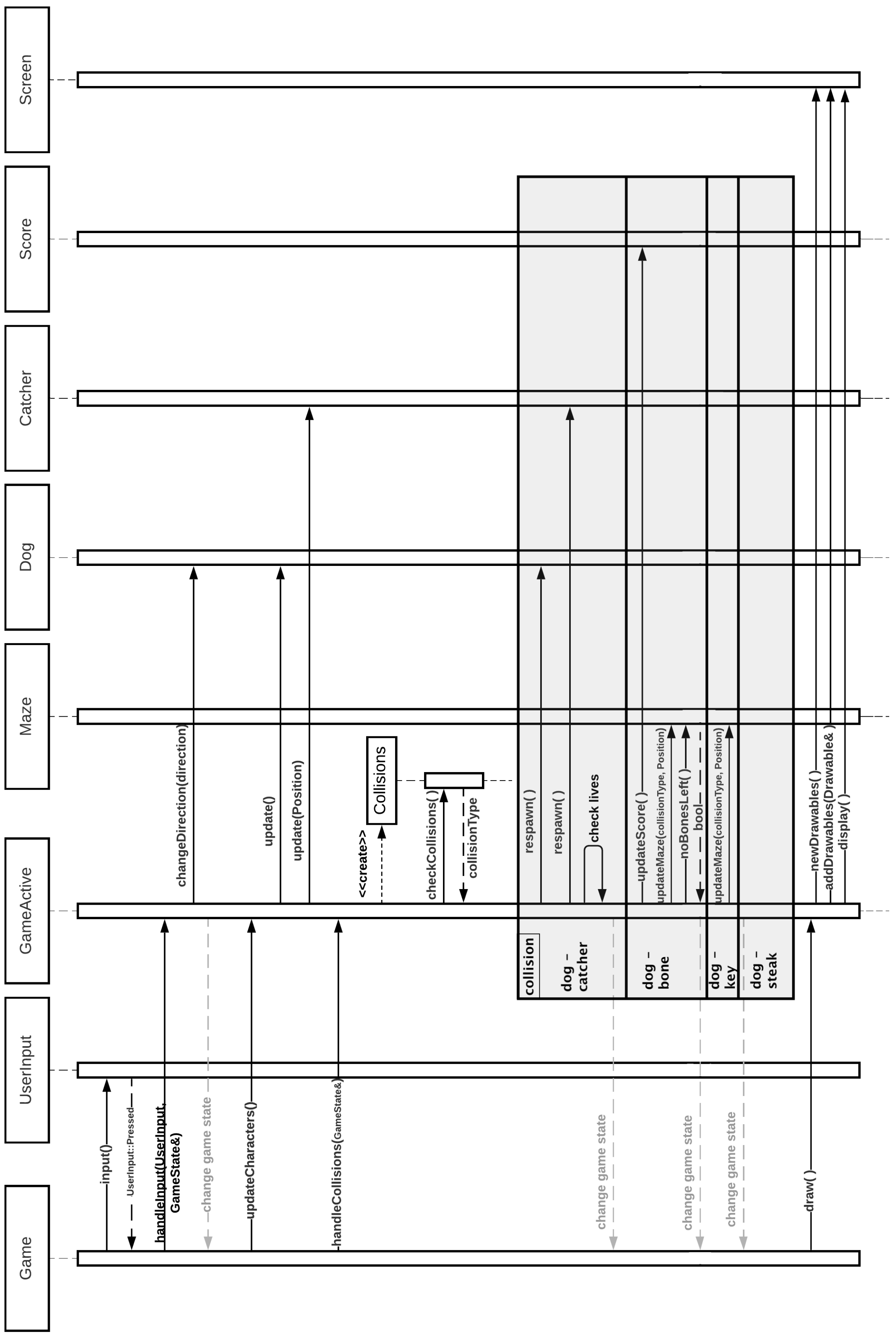

Figure A.1: Sequence diagram when the game is in an active state. The sequence here represents a single loop in `Game`'s `run` function. Grey return arrows have been used to represent where the state of the game may change from active.