ELEN3009 – Software 2 Project Report, Group 1, Super Pac-Man

Duncan Smale 1619539 **Jonathan Taylor 1665909**
School of Electrical & Information Engineering, University of Witwatersrand, Johannesburg, South Africa

Abstract: This report discusses the design and implementation of a Super Pac-Man clone. The high level design decisions and complex algorithms are explained and justified. The core functionality of the game is present with the addition of all major features and most minor features. The project was a success with good design decisions, however some classes require improvement if the project were taken further.

I. Introduction

This report follows the design and implementation of a Super Pac-Man game clone. This project is written in C++ and makes use of the SFML 2.5 graphics library to handle object rendering. **Section II** identifies the key areas of focus within the project. **Section III** explains the Domain Model of the project. **Section IV** expands on the code structure and class hierarchies while explaining the design choices made when creating the classes. **Section V** provides a critique of the overall design and future improvements. **Section VI** identifies classes which were not tested and provides insight as to why.

II. Problem Identification

The task given is to create a clone of the game Super Pac-Man. The given scenario allows a large degree of creative freedom in the implementation and aesthetics of the final product.

The final design is oriented around four main layers of responsibility. The Logic/Management layer, the Utility layer, the Application layer, and the Presentation layer. The project is structured around maintaining these layers respective responsibilities by splitting the game logic and behaviours into small classes.

The key classes which will be explained are the maze class and its hierarchy. It is responsible for maintaining the grid representation of the map. The gamemanager and gamestate classes are important for separating the game logic and phases into separate classes. The pacman and ghosts class are important as they form the core gameplay loop of movement and collection.

III. Domain Model

The Domain Model of the project encmpasses all classes and their relationships with one another. The gamemanager is the most connected class as seen in Figure 1. The gamemanager is used elsewhere in the code to reference the pacman and ghosts instances, however to represent this in the Domain Model create too many relational lines.

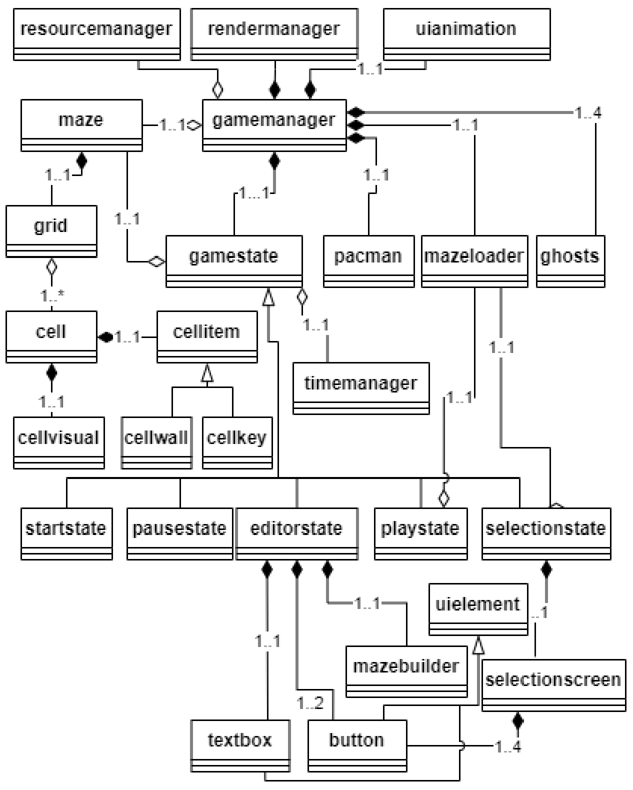

Figure 1: Domain Model

The domain of the project was modelled to have a central logic structure which would direct the game as necessary. The central structure would then create and assign the necessary objects required for that particular phase of the game. These objects would be of the Application and Presentation layer which would generate the game functionality.

The maze, grid, cell, and cellitem relationships form a core part of the gameplay loop. This relationship is accessed through the instance of the maze class created within the gamemanager. This instance is then passed where needed through static memory by the gamemanager or upon the creation of the object such as is the case in the gamestate and its derived classes.

IV. Code structure

There are four main layers within the project: Logic/Management layer, Application layer, Utility layer, and Presentation layer. Most of the Application layer contain visuals which are then rendered by using a class in the Presentation layer, namely the rendermanager, however they remain in the Application layer as their primary concern is generating game behaviour.

The Logic/Management layer is made up of the gamemanager and the gamestate. This layer is focused on the state the game is in and when the state should change.

The Utility layer is made up of the resourcemanager and the timemanager. These classes provide useful data and resources to the rest of the project.

The Application layer is made up of the following classes: maze, grid, cell, cellitem, mazebuilder, grid, cell, cellitem, maze, pacman and maze ghosts'. These classes are responsible for generating gameplay and game behaviours when interacted with either directly, or indirectly.

The Presentation layer is made up of the rendermanager, cellvisuals, uielements. This includes uielement, selectionscreen, button, textbox, messagwindow, uianimation, and score. These classes are mainly visual and UI focused classes that give the user visual information. The selectionscreen, button, and textbox do generate a minimal amount of gameplay behaviour and could be considered as partially part of the Application layer, but this is only when directly interacted with and for short periods of time.

Classes and Class Hierarchy

The main class hierarchies that exist within the project are split by their layer.

The Logic/management layer contains one hierarchy. This is made up of the gamemanager and gamestate hierarchy. The gamemanager makes use of the gamestate class to create game logic each frame.

The Utility layer contains no strict hierarchy and is used as a distribution mechanism for resources needed throughout the project. The timemanager distributes the frame time, and the resourcemanager image resources required.

The Application layer has multiple hierarchies. They are as follows:

1. maze, grid, cell, cellitem, cellvisual.
2. mazebuilder, cell, cellitem, cellvisual.
3. maze, pacman, cell
4. maze, ghost, cell

The Presentation layer has smaller hierarchies due to the UI components being independent of one another. But they can be seen as follows:

1. rendermanager, selectionscreen, button
2. rendermanager, button
3. rendermanager, textbox
4. rendermanager

The uielements class is the base class for many of the UI components and would be included in the hierarchies if it provided functionality as opposed to defining the public interface.

1. gamemanager and gamestate

The gamemanager is modelled to be the main controller of the game. This class is responsible for initiating the game loop and running the game loop every frame. This class manages the high-level game logic. This was done to ensure the class was not too long and in control of too much game logic. The lower level game logic and game state is offloaded to the gamestate class and its derived classes. The gamemanager is responsible for reading in user input and changing state depending on what was pressed. The gamemanager is responsible for the transition between gamestates. The gamemanager makes use of the uianimation class to create smooth gamestate transitions

As the gamemanager is to manage the higher-level game logic and user input, the gamestate class is modelled as a state of play. A state of play can be entered to prepare the necessary resources, activated to produce game logic, and exited to release memory and remove visuals. The gamestate base class is used as interface inheritance for each of the inherited classes. The inherited classes model each of the states the game can be in, which are startstate, selectionstate, playingstate, editorsate, and pausestate. These states were chosen as they directly map to each phase the game can be in. This breaks the game logic into much smaller and manageable components which will be run every frame by the gamemanager.

The startstate is responsible for displaying the splash screen and game controls.

The selectionstate is responsible for the user selection of the map which is facilitated by the selectionscreen class. This state retrieves the map to be loaded from the selectionscreen and makes it available by loading it into the gamemanager. This is to ensure that the map name is available to all other gamestates.

The playingstate is responsible for loading the selected map on entering and updating and rendering pacman and the ghosts every frame.

The pausestate is responsible for ensuring pacman and the ghosts are rendered, but not updated.

The editorstate encapsulates the logic required to use the mazebuilder class and is responsible for monitoring mouse input and mouse location required to build a maze. This design allows the gamemanager to become much more concise and encapsulate the key behaviours of each state of the game.

2. timemanager and resourcemanager

The timemanager is modelled as a time holder able to distribute the time value wherever needed. It is the responsibility of the timemanager to store the given time. The timemanager is responsible for being able to distribute the time it holds to any place that requires it. As such the timemanager is accessed through static functions such that the deltatime for each frame can be accessed anywhere it is required.

The resourcemanager is modelled as a collection point for game resources used throughout the project. The resourcemanager acts as a static class. This is to ensure that the resources are only loaded once, and the same resource will be distributed as required. The resourcemanager is responsible for loading the resources into static memory.

3. Game Behaviour Hierarchies

The maze is the overarching class in this hierarchy, responsible for acting as the interface between the grid class and other objects while also being responsible for the path finding algorithm for the ghost class. The maze class was modelled to turn the grid class, which is a grid representation, into a maze where direction and movement is important. This choice was made to separate the state of the grid from the gameobjects that make use of the grid. The maze makes use of the Breadth-First-Search (BFS) algorithm for Ghost path finding. This was chosen as graph theory can be applied to a grid representation and provided a neat and accurate path finding tool which when given different end cells can provide different paths for the ghosts to follow.

The grid class is modelled to represent the map in a two-dimension grid format where each space is considered a cell, each cell has a width, a height, and a value. The grid is responsible for creating the maze. Furthermore, the grid is responsible for basic map queries of movement. The grid manages the map as a two-dimensional vector of cells and is responsible for handling the score call-back binding through C++'s functional library. The functional library was used as a function can be bound at run time which easily decouples the cell class from all other classes to maintain independent state. The grid class further extends its management of each cell by assigning the visuals of the cell based on the value of the cell. This was done to reduce the responsibilities and complexity of the cell class as the grid class has access to all the cells and their positions in the grid. The grid class is responsible for giving each cell its world position and size on construction. Another responsibility of the grid class is setting up the key and door links. The door locations are stored as a two-dimension vector of int pairs called keylocations, the pair of int's state the row and column on the grid the door is located at. Each key is then assigned the cellitems from the keylocations sub-vector which corresponds to their index in the keys vector. The algorithm used to create the key and door links is described by Figure 2. This algorithm is present in the grid class as it has access to all the cells and their items. This was implemented as such to decouple the state of each cellitem from the rest of the grid.

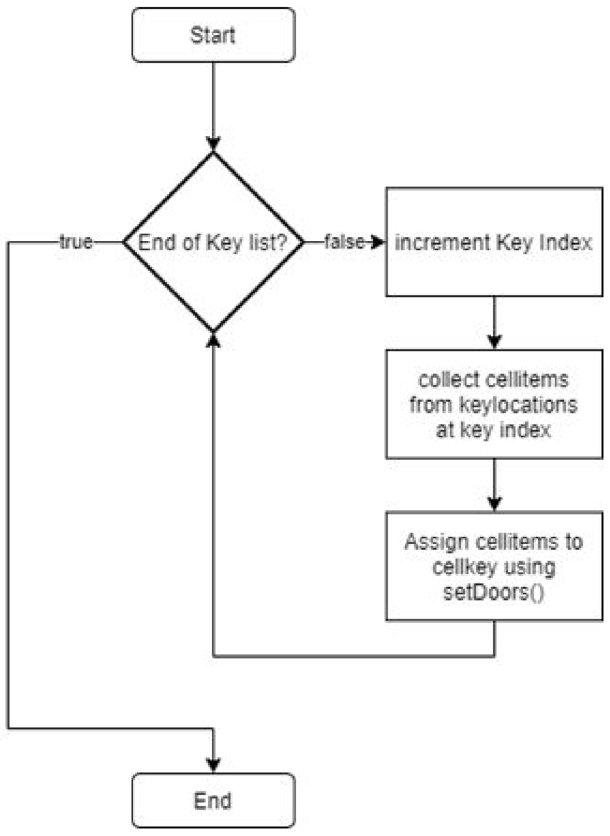
Start
End of Key list?
true
false
increment Key Index
collect cellitems from keylocations at key index
Assign cellitems to cellkey using setDoors()
End

Figure 2: Key/Door Assignment algorithm

A cell is modelled as a region of space on the screen with an item inside. The cells class is responsible for the item in the cell, the score added once picked up, and changing the visuals of the cell once picked up. A cell is used as a measure of movement for the game, this is done to simplify the movement system within the game by moving towards a cell centre, as opposed to just in a direction. Each cell is given a value on construction in the form of a char. The value is used to determine which cellitem derived class will be used through a switch statement. Pacman and the Ghosts use cells as movement waypoints by moving to the cell's centre position. The cell is responsible for changing its internal state once Pacman reaches the centre of a cell. Once Pacman has reached the centre of a cell, the cell is to become open and all other states are to be cleaned up. This is handled through changing the current item in the cell to the base class cellitem.

A cellitem is modelled as an object that can be placed in a cell. A cellitem can be activated and should modify its parent based on what the item is. The cellitem class is a base class where all items inherit from it. There are two derived classes: cellwall and cellkey. There are only two sub classes as the behaviour of the other classes corresponds to what the cellitem does inherently. The different behaviour exhibited by the cellwall, which does nothing when activated, and the cellkey which activates a set of doors. The cellitem is a vital class when defining game logic and behaviour. This class handles the removal of fruit, keys, doors, power, and super pellets upon activation. The most important activation interaction is the cellkey. Each cell key has a vector of cellitem's that will be activated when the key is activated, each of these correlates to a door.

Another class which is linked to this hierarchy is the mazebuilder class. The mazebuilder is modelled as a user interface where each cell is outlined by gridlines and each cell can be clicked on. When clicked, the value will be altered based on what mouse button was used, Figure 3. This class is a friend of the grid class, such that it can modify its internal state which is necessary for the creation of new maps as adding public functions to the grid class could lead to unwanted changes to the grid while playing. This class is responsible for handling user mouse input: left, middle, and right click. Each click performs a different action when building a maze from the UI. Different clicks placing different cells allows the user to quickly build a map.

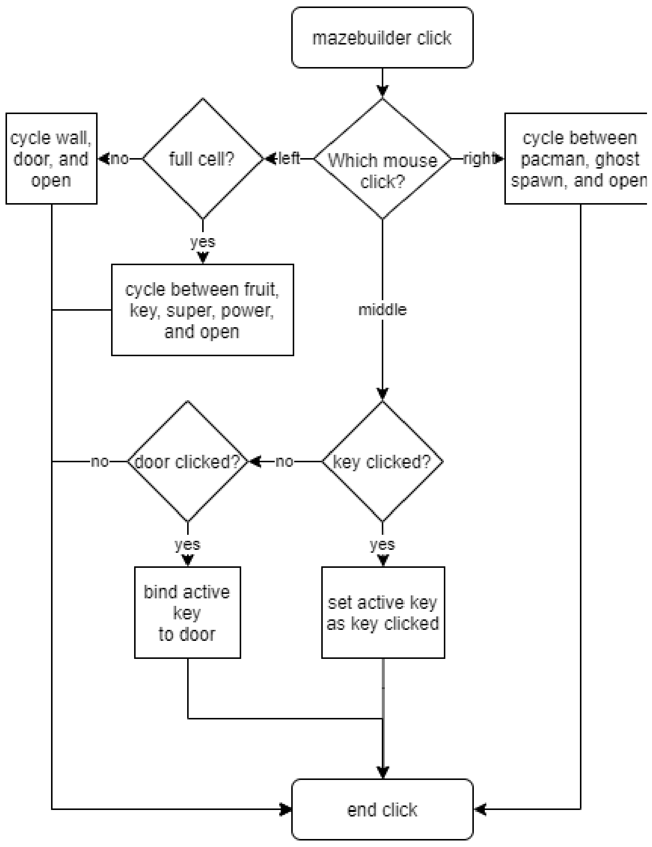
mazebuilder click
Which mouse click?
left
right
middle
full cell?
no
yes
cycle wall, door, and open
cycle between fruit, key, super, power, and open
cycle between pacman, ghost spawn, and open
key clicked?
door clicked?
no
yes
set active key as key clicked
bind active key to door
end click

Figure 3: mazebuilder click behaviour

The mazeloader is modelled as a read and write system. The mazeloader is an important class which is responsible for the loading and saving of maps into files. This class parses the files into the two-dimension char vectors and ensures the key/door pairs are linked and saved correctly. This class is also responsible for saving and loading the valid maps that have been created by the user. Each valid map is saved as a string and int pairing, the string representing the map name and the int representing the current high score.

Pacman and the ghosts classes derived from Entity. Entity contains basic functionality shared between all moving characters in the game. These entities are declared and initialised in gamemanager. Entity contains many getters and setters, mainly used in gamestate for updating parameters of the entities and for testing.

Pacman receives user input and moves pacman in that direction until it hits a wall or the direction is changed. Only essential methods and getters were made public in pacman such as moving pacman and if pacman has the super pellet or not, the rest were accessed privately in the move function of pacman.

Ghosts are enemies of Pacman and will chase Pacman through the maze using the Breadth First Search discussed earlier. Only essential methods were available publicly, all other updating methods such as the updating the random directions when scattering were kept private and accessed by the moving ghost functions, which essentially acted as update functions.

4. UI Hierarchies

The rendermanager is modelled as a queue of items which need to be rendered each frame. An item can be added to the render queue, and subsequently removed when needed. The rendermanager should be available everywhere. The rendermanager's responsibilities include drawing the items in the queue every frame. The rendermanager uses a vector of drawables and an int to use as its priority. This is to ensure that the higher priority items (larger int priority) get rendered above the other items. Each time an item gets added to the render queue, it is sorted to ensure the item is placed correctly. The rendermanager makes use of the singleton pattern to ensure only one instance exists. This ensures we only add to one render loop and any object can be added to the loop.

The cellvisual class is used to display the visuals for any cell or object. The class is only used to display visuals of a cell and holds minimal state, namely its current texture and the position on the texture where the visual for the sprite will be found. The position of the visual is also used, as well as the size of the sprite. This class design allows for dynamic assignment such that it can be used by any type of cell.

UI forms a vital part of the gameplay loop. The most prominent of these is the uielement. This class is the base class for all UI components that are present within the game. This class is responsible for defining the public interface with which all UI uses. It is modelled such that a uielement will change state when a user mouses over an element. A uielement has a region it can be clicked on. A uielement needs to be checked if it was clicked on, and a uielement handles its own state change when clicked on. A uielement and all its derived classes are responsible for maintaining its own position and providing a public method to update the position. A uielement provides public methods to render the uielement and remove the uielement from the render loop. The derived classes will use the rendermanager to handle what child elements will be rendered, and in what order.

A well utilised UI class is the button class. This class uses the interface defined in the uielement class and extends them to define button behaviour. A button is modelled as a clickable entity which will perform an action when clicked on. The button class can be in one of three states: idle, hovered, or clicked. The button changes background colour based on what state the button is in. This is to provide user feedback. The button is responsible for generating a call-back when it is clicked on. The call-back system uses the C++ functional library to bind function pointers. A call-back feature was used as it can be assigned dynamically, thus allowing buttons to be used throughout the project.

The main UI class which is interfaced with is the selectionscreen. The selectionscreen is modelled as a UI panel where the user can change the selected map, view the high score and select whether to edit or play the selected map. This class is responsible for changing the selected map and allowing the player to edit or play the selected map. The selected map can be requested from the selectionscreen and is used to update the selected map for use elsewhere in the project. This was done to ensure the user input logic for selection was encapsulated in one class.

A textbox is modelled as a region of space where a user can click and enter text. The textbox class is used once within the project. This class is responsible for getting user input in the form of a string and displaying it. The class is derived from uielement and makes use of the public interface. This class is only active when selected, and as such changes colour when selected to give feedback to the user. This class is responsible for inputting the map name during the editor state for when the player wishes to save their map.

The uianimation class is used as a basic fade in and fade out transition mechanism. It is modelled as a fade-able rectangle which will be used to transition from one screen to another. This class is responsible for creating a smooth transition between gamestates, such that the user is not jarred by the sudden change of visual. This class uses call-backs to send messages when each of the transition states is complete. This ensures that the gamestate can be exited, and the new state entered while the screen is dark, and the objects loaded behind.

Displaying messages to the user forms a vital part of the game loop. The messagewindow is modelled as a feedback window that displays a message when requested. The window should pause the game upon start and resume again upon close. The messagewindow class is responsible for displaying these messages in their own RenderWindow. This was chosen as a call-back can be generated when the RenderWindow closes.

It was required that a score system be put into the game. As such the score class was implemented. The score class models a traditional scoreboard, where the score can be incremented, the user can get the score and the score will be displayed to the user. The score class is responsible for displaying the score to the player. The score class is further responsible for incrementing the score when requested. This ensures that the score system is only responsible for current score, and is decoupled from the rest of the project code.

5. Dynamic Object Collaborations

The main dynamic object collaborations are made up of two game layers. The primary collaboration is between the gamemanager and the current gamestate as illustrated in Figure 3 below. This figure displays the logic that the gamemanager processes when it receives a call-back or proceeds to the next frame.The gamemanager makes use of two possible call-back phases, the uianimation complete call-back, and the transition to next state call-back. The uianimation call-back is assigned when the animation begins. Whereas the next state call-backs are assigned when the gamemanager is created. Each gamestate has its own function call-back assigned within the gamemanager to transition it to the correct next state.

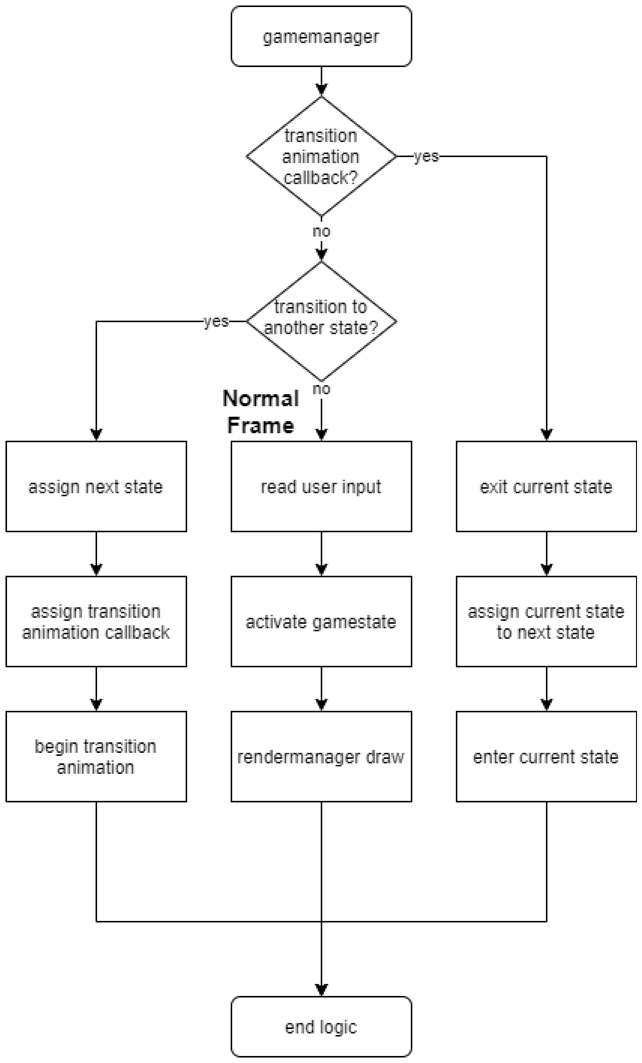
gamemanager
transition animation callback?
yes
no
transition to another state?
yes
no
Normal Frame
assign next state
assign transition animation callback
begin transition animation
read user input
activate gamestate
rendermanager draw
exit current state
assign current state to next state
enter current state
end logic

Figure 3: gamemanager object collaborations

The second collaboration is between the maze hierarchy, pacman and the ghosts. This collaboration happens in gamestate, where every frame, it is checked if pacman reaches the centre of the next cell. If he does, various conditions are checked and updated that alter the score and even the mechanics of the game with pellets, this is logic is illustrated in Figure 4 below.

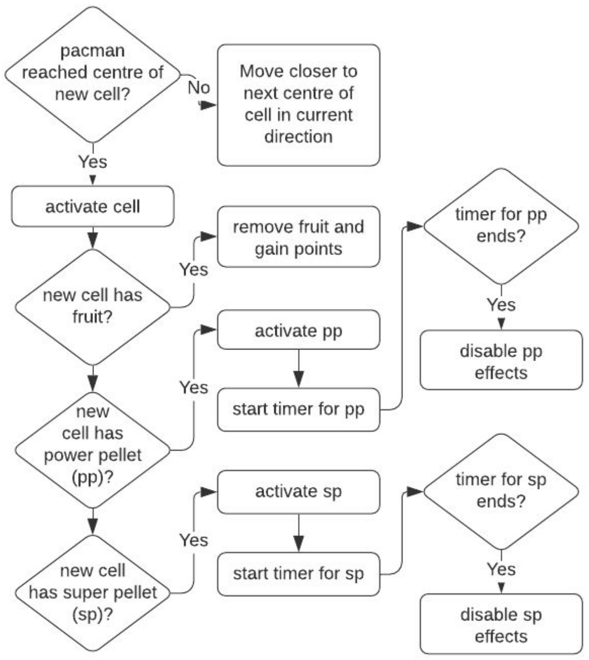

Figure 4: maze object collaborations

Every frame, gamestate also checks if pacman has collided with a ghost under the conditions seen is Figure 5 below.

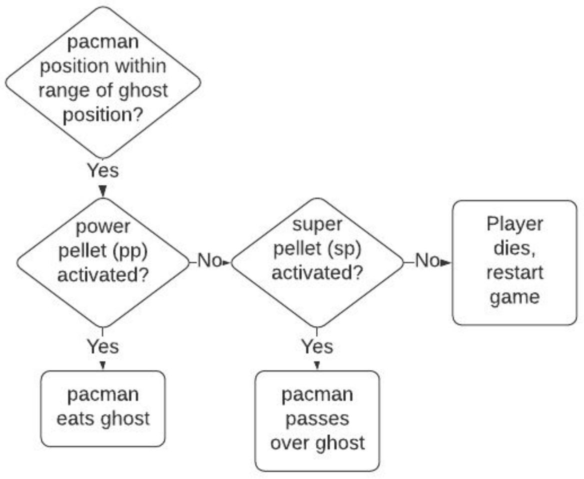
pacman position within range of ghost position?
Yes
power pellet (pp) activated?
No
super pellet (sp) activated?
No
Player dies, restart game
Yes
pacman eats ghost
Yes
pacman passes over ghost

Figure 5: entity object collaborations

V. Design Critique

The overall architecture of the project was well structured with a clear distinction of state logic, game behaviour and game presentation. The gamemanger and gamestate hierarchy and relationship provided a much needed game logic separation between the phases of the game. The gamestates allowed for easy extension of game logic by adding another derived class of gamestate. There are some issues in the design with the maze and grid relationship. Further design issues arise when viewing the gamemanager class and how the instances and public variables are managed.

The maze and grid design represent and model the same idea. The logic of the grid should have been simplified to cell manipulation, while the maze was given the movement query logic. This would have further reinforced the idea that the maze represented the combined area of play, while the grid acted as the holding mechanism of the cell.

The gamemanager primarily uses static functions and static member variables to ensure that the required objects are accessible anywhere within the project. This poses a distinct problem where the gamemanager class is then tightly coupled to a large portion of the code base. A solution to this would be better use of dependency injection within the constructor of certain objects outside of the Logic layer. This would remove the need for calls to the gamemanager namespace as the objects would have the necessary dependencies resolved on creation. In the case where the objects are dynamic, the class should be responsible for providing the Logic layer a mechanism to alter the dependency as required by the game phase. An alternative approach but which still maintains the coupling issue would be to make the gamemanager class a singleton. This would ensure only one gamemanager exists at a time.

The cell design should have been abstracted further to define specific cell types using an enum class as opposed to continuously using a char value. The char value is not good design as the value could be mistaken somewhere in the codebase. An enum class would provide a concrete data type to use and would throw an exception or be easily traced where the error occurred.

The grid class contains too much information regarding the cell visuals and creating the cell visuals themselves. This functionality should be moved into the cellvisuals class or otherwise refactored into its own class to maintain strict class responsibilities.

The game possesses all required functionality except for random spawning of stars to collect and multiple lives. The functionality is broken down into their individual classes and behaviours. The visual fidelity of the project could have been improved through using sprites for the buttons, however due to time constraints this was not implemented.

A further improvement would be to introduce sound effects and background music to the game. This would improve user experience and enjoyment of play. For this to occur, a sound manager would need to be implemented which might increase the coupling of classes. This would require much greater planning in future iterations.

Entity could have been made an Interface as only derived classes (pacman and ghost) are initialised in this project. Additionally, pellet could have been made a base class, with power and super pellet deriving from pellet. Movement of pacman could have been made a separate class to pacman and pacman could be composed of it.

In addition, many checks, such as being at the centre of a cell, in pacman and ghost were called every frame. This process is similar to polling, instead of a more efficient solution - only being reacted to when a parameter changes, similar to an interrupt.

VI. Testing

There are a few classes which are not tested within the project. The classes are as follows: gamemanager, all gamestate derived classes, messagewindow, mazebuilder. Some classes were only partially tested, such as the selectionscreen.

The gamemanager and gamestate derived classes were not tested due to them being the classes responsible for creating the game flow. The game flow is impossible to test within this project without having user input.

The messagewindow class was not tested as it is purely a message display system. This would require a use to be present to close the messagewindow for the rest of the tests to be completed.

The mazebuilder was not tested as it is purely driven by user input and mouse clicks. To test this class multiple test user inputs would need to be created and separate functions used to query the state of a cell. The inclusion of these functions would only serve a testing purpose and would not add any value to the project and thus are not included.

The selectionscreen is only partially tested as this class is user input driven, with button bindings necessary to complete the tests. As the class is mainly UI, user input would need to be generated to simulate clicking a button, but this would require far too much internal information about the state of the class. This type of testing would add no value to the tests and thus is not fully tested.

The entities' parameters such as direction, position and texture were tested, including ghost's random direction used in scattering of ghosts. Pacman and the ghosts movements were tested. The process of reaching a cell centre (seen in Figure 3) and updating the game based on the outcome was tested. The interaction between the ghosts in pacman (as in Figure 4) was only partially tested. This interaction proved difficult to test as it involved many different classes interacting together, being pacman, ghost, gamestate and maze. The timers for the pellets were tested indirectly with the setting of the deltaTime variable.

VII. Conclusion

The overall project was a success. All the required functionality is present in the final iteration. The project includes all major features requested and all the minor features except for random star spawns and multiple lives.

The project would further be improved by limiting the dependencies between classes by using a form of dependency injection on construction. The maze and grid class relationship should be improved to distinguish each class' responsibility within the project. Entity's derived classes could involve more interrupts, instead of polling conditions.