Bubble Trouble - Group 4

Group members: Asif, Darshan, Navin, and Sighvatur

Aim: To create a bubble trouble game using C++

Table of Content

Recap of Cycle 1 implementation:

In this iteration, the following game dynamics are implemented.

1. Once a playing session is started, a player can move either left or right using “A”, “D” keys respectively.
2. “Spacebar” key is used to shoot the spike.
3. A Spike can be shot one at a time until it either hits the ball or hits the topmost point of the frame.
4. There are different sizes of the ball with different colors. The balls have different velocities.
5. Once a spike hits a ball, it splits into two smaller sized balls.
6. The new balls created attains new velocity.
7. The frame rate is set at 60 fps.

Cycle 2 implementation:

In this iteration, the following game dynamics are implemented.

8. Once a playing session is started, a timer starts.
9. A player is given three lives initially.
10. If the player collides with a bubble, he loses one life, the stage is restarted so the timer and score are reset.
11. A sound plays if the player collides with a bubble.
12. The score is increased by 1 when the player pops a bubble.
13. Fixed bubble types are created, with fixed properties so they move more predictably.
14. If a bubble is burst, an explosion image is shown, and a sound is played.
15. While the spike is shooting, a sound is played.
16. The game can be paused or un-paused by pressing the “p” key.
17. When the player is out of lives, the game is paused.
18. While the game is paused, pressing the Enter/Return key will restart the game.
19. Pressing Escape at any time will quit the game.

Recap of cycle 1 Design Decisions:

- A game can have an engine which is responsible for rendering of the game, initializing game dynamic components, updating as the game progresses and cleaning of the objects when the game has ended/exited. Hence, we have a “GameEngine.h” which on a high level is responsible for the specifics mentioned earlier.
- A game has many components and may communicate with various other classes to get the required objects. So, templating of C++ comes in handy. Hence “GameObject.h” does tracking of

the newly added components/objects being added to the list. This exhibits code modularity and reusability.

- Also, the components need to be initialized and updated as per the events taking place in a game, having virtual functions to handle the corresponding events is wise. So, a class called "GameComponent" with the GameObject class is placed. The virtual functions are overridden in the respective game handler class/events. Most of the classes extend the "GameComponent" class. This indeed reduces code duplicity and code redundancy.
- For the current implementation, the game can be played via a keyboard. The pressing of buttons forms an event. Hence a dedicated class called "KeyboardHandler" has been made to handle the keypress events. When a player presses the dedicated keys, C++ knows the events and the functionality assigned to those events. Of course, SDL libraries come in handy here.
- In the game, various movements are observed
    - A player moves either to right or left.
    - Balls created are moved randomly in the 2D plane.
    - Once a ball is shot, it splits and becomes a smaller size from the parent ball size and attains certain velocity.
    - If the balls are in the same plane as the player, if yes, the player loses a life (will be implemented in the later cycles.)

  Keeping track of the positions of above-mentioned scenarios are important and the checks must be made in certain scenarios. Hence a dedicated class to check the collisions is tracked under "CollisionChecks". Now, once the checks are made, "CollisionHandler" class will handle the events like splitting the balls into two parts and assigning the respective velocity based on its updated size (will be less from the parent ball size).
- As mentioned earlier, all the objects' position, velocity, and acceleration are calculated. "MovementHandler" class handles tracking such parameters during the game.
- Textures for the player, balls, and spike must be loaded and rendered. Hence, a separate class to handle rendering of the different texture called "TextureLoader" class.
- To render spike and the player, we have a class called "TileHandler" class. This handler will handle events like loading the texture of player, balls, and spike (uses TextureHandler) and scales the respective images accordingly.
- loaded and rendered. Hence, a separate class to handle rendering of the different texture called "TextureLoader" class.
- In order to render spike and the player, we have a class called "TileHandler" class. This handler will handle events like loading the texture of player, balls, and spike (calls texturehandler) and scales the respective images accordingly.

Cycle 2 Design Decisions

Design decisions made in this cycle also reflects some of the alternate changes that have been made to the cycle 1 design.

- Previously, all the objects in the game used the GameObject class. Now, each GameObject extends from its own object type. For example, a player has his own object called “PlayerObject” which extends the GameObject. It’s true for all the other elements in the game, which will be explained in the upcoming sections. The advantage doing this is that object-specific code can be made for a constructor, update etc., but there is more code and adding an extra object is more difficult. Currently, only the bubble and player object implement different functionality, but the additional classes are necessary for the ObjectManager class.
- The ObjectManager class was added to keep track of each type of object in an array of vector. What this means, is that we can just add objects however we want, and they will get updated, drawn etc., so there is less necessary update call code and it’s easy to iterate over each object type. Another advantage this gives us is that adding a new object type is not difficult, but the current implementation uses templating of the GameEngine base type, so each object needs its own class, as mentioned above.
- A feature in the game is to pause/unpause game or restart based on the keyboard keys pressed by the player. To handle the timing of the game, a dedicated class called “MillisTimer.h” is available and the associated functions can be reused whenever necessary. For example, for timed stage play. (Will be implemented in future)
- The “BubbleObject” class creates the bubbles and randomizes their properties, in order to make the game challenging. The random values are obtained from the “Randominterface” class. All the above explanations are depicted in the form of class diagrams and inheritance diagrams.

Recap of cycle 1 class explanations

Following are the classes involved to achieve the above-mentioned implementations.

1. Main.cpp
    a. Entry point to the game
    b. Responsible for initiating the Game engine, window size, position.
    c. Runs the main game loop, calls handleEvents, update, cleanObject and render functions of the Game.
    d. Locks the framerate at 60fps as most of the modern computers have a screen refresh rate of 60Hz.
2. GameEngine.h
    a. Handles all game logic by initiating the objects, window, and background, handling events, updating all objects, cleaning them and calling draw.
    b. The event handlers, SDL window, and objects for the player, spikes are created here.
    c. The boolean to initiate the Game isrunning() is provided from this file.
    d. The GameEngine starts the game and cleans up the objects when the game has ended.
    e. The tile is rendered for both bubbles and the player.
    f. A color array is given to change the colors of each bubble created.
    g. The update for the player movement and spikes are created here.
    h. Uses functions from Collisionchecks.h to decide to destroy the spikes and bubbles.
3. Gameobject.h

   a. Includes the component base class of GameComponent in order to use it for templating.
   b. Has an interface to allow components to be added and accessed, using templates.
   c. Iterates over each component for each update and draw call.
4. CollisionHandler.h
   a. Extends the GameComponent class
   b. In each update call, it checks if the object associated with it is colliding with the playing zone, and changes its position and velocity if appropriate.
5. KeyboardHandler.h
   a. Extends the GameComponent class
   b. Keeps track of the keyboard events, updating velocity and enabling the spike object.
   c. Implements “AD” keys for left and right movement respectively.
   d. Also “left-right arrow” keys for left and right movement respectively.
   e. “Spacebar” keys for shooting the spike.
   f. Velocity flag is flipped accordingly to the movement of the player i,e, velocity is positive towards right movement and becomes negative to move towards left.
   g. SDL_GetKeyboardState is used to determine what keys are being pressed.
6. TextureLoader.h
   a. Loads the textures of the game by making use of the SDL image library and SDL_CreateTextureFromSurface method.
   b. Applies color to the texture, used for the bubbles.
7. MovementHandler.h
   a. Extends the GameComponent class
   b. The position and velocity of all the game object are maintained and updated here.
8. TileHandler.h
   a. Handles loading and rendering of textures using TextureLoader.
   b. If a texture is loaded, the texture is destroyed once the game has ended.
9. Collisionchecks.h
   a. The boundaries of both the player and the bubbles are checked.
   b. The functions check if any of the game objects collide with one another at the point or not.
   c. The function collideswithrect checks between the rectangular background, spike, and the player.
   d. The function collideswithcircle checks between all the game objects.

Cycle 2 class explanations

10. SoundHandler.h
    a. Responsible for playing the necessary sounds in the game like
       i. When the bubbles collide.
       ii. When the spike is shot.
       iii. When a bubble collides with the player/ when the player loses a life.
    b. Makes use of SDL_Mixer library, which functions

        i. Opening audio channels (mono, stereo), setting the audio frequency and sampling rate.
    c. Freeing the memory when the audio isn't playing.
11. PlayerObject.h
    a. Inherits from the GameObject class.
    b. Keeps track of the accumulated score for the stage.
12. BubbleObject.h
    a. Inherits from the GameObject class.
    b. Custom constructor which initializes all the game components necessary, with the correct parameters.
    c. Added function for getting the next bubbles in case of a pop.
13. ExplosionObject.h
    a. Inherits from the GameObject class.
    b. Updates all the components associated with the initialized explosionobject.
14. FontObject.h
    a. Inherits from the GameObject.
    b. Responsible for the creation of a texture from inputted text and rendering.
        i. Setting the position of the texts.
        ii. Rendering the texts.
    c. Makes use of SDL_ttf library.
15. MillisTimer.h
    a. Responsible for tracking the time passed as the game progresses.
    b. Has pause and unpause functionality.

Cycle 2 Evaluation

The GameObject class and components are working well, but we would like to have a better way of checking for collision and doing specific things depending on which object is what.
Additionally, the new ObjectManager could be improved, by not needing to create a separate class for each gameObject, we need to add an enumeration feature instead.
We have some ideas on how to set up different levels and the menu, by using a state machine for the game, we could clear the screen and all objects when jumping to a new state, and add the correct objects as appropriate.

Of note, is that we have encountered a segfault error which only appears on some Linux machines. We have not been able to fix it before delivery. The error seems to have its roots in the FontObject class in the setText function.

Game description

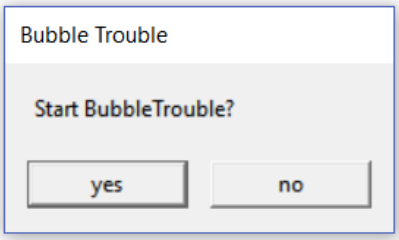

Figure-1

Figure 1 is popped up when the user starts the game, selecting yes will start the game.

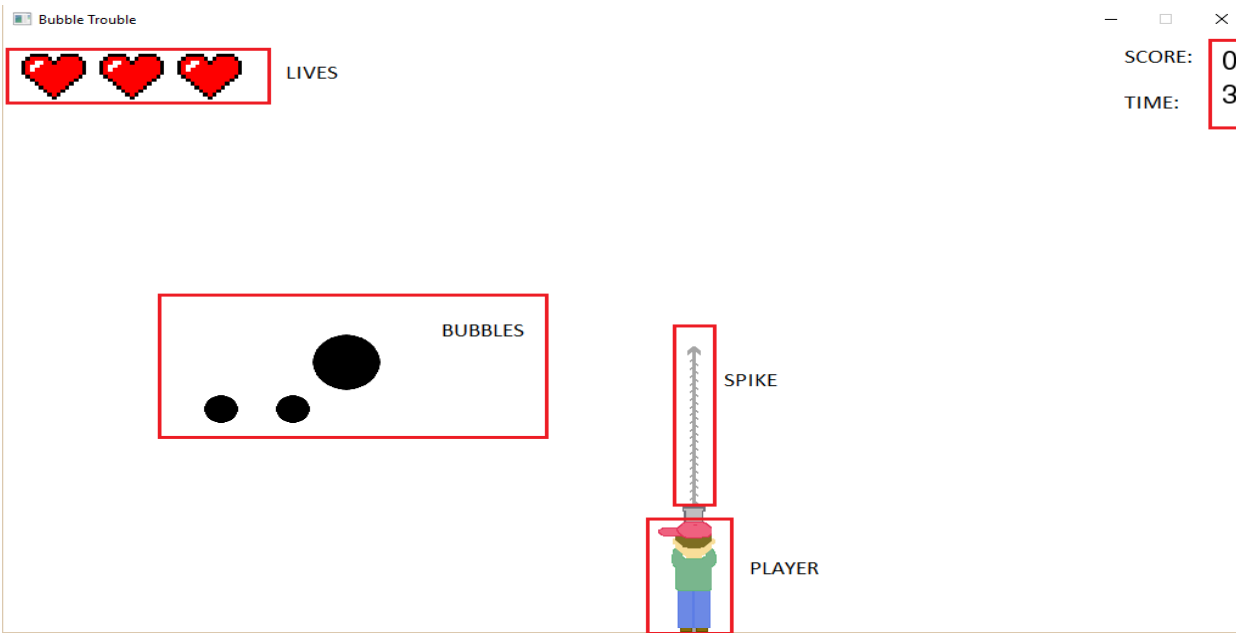

Figure-2

As soon as the player enters the game, **figure 2** window is seen. The window contains

- A player
- Lives
- Score
- Time

Pressing "Space Bar" will enable a player to shoot a "spike" as seen in the figure 2. As soon as the spike hits the top of the play area, it disappears.

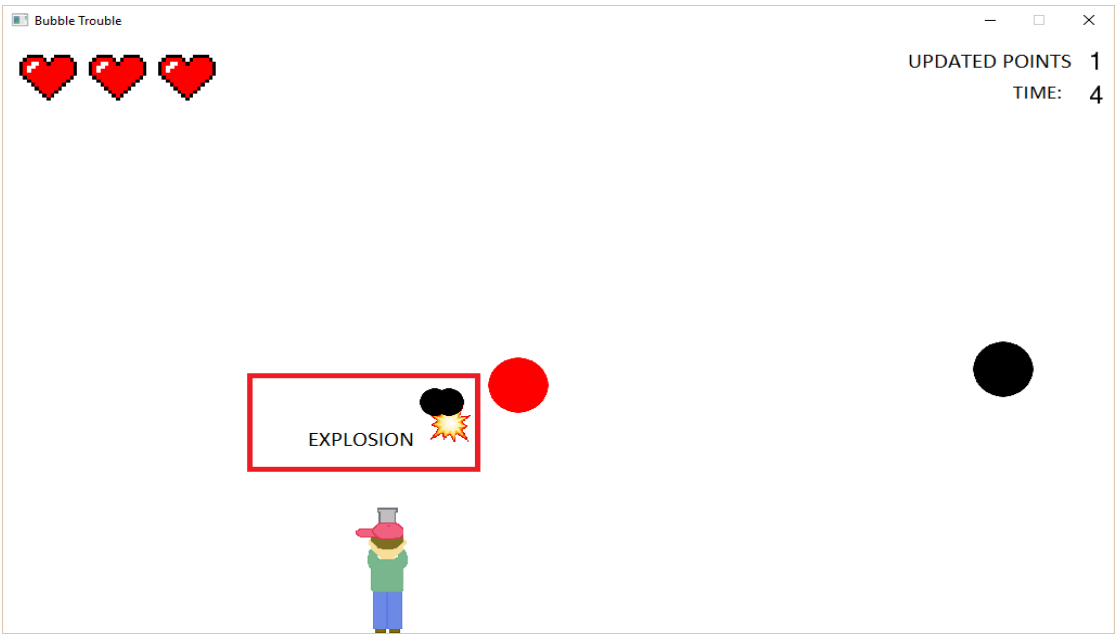

Figure-3

Figure 3 shows that when a player hits a bubble, the bubble splits and explosion sound is heard with a small explosion image around the newly created bubbles. As a result, the score is updated and the spike is destroyed.

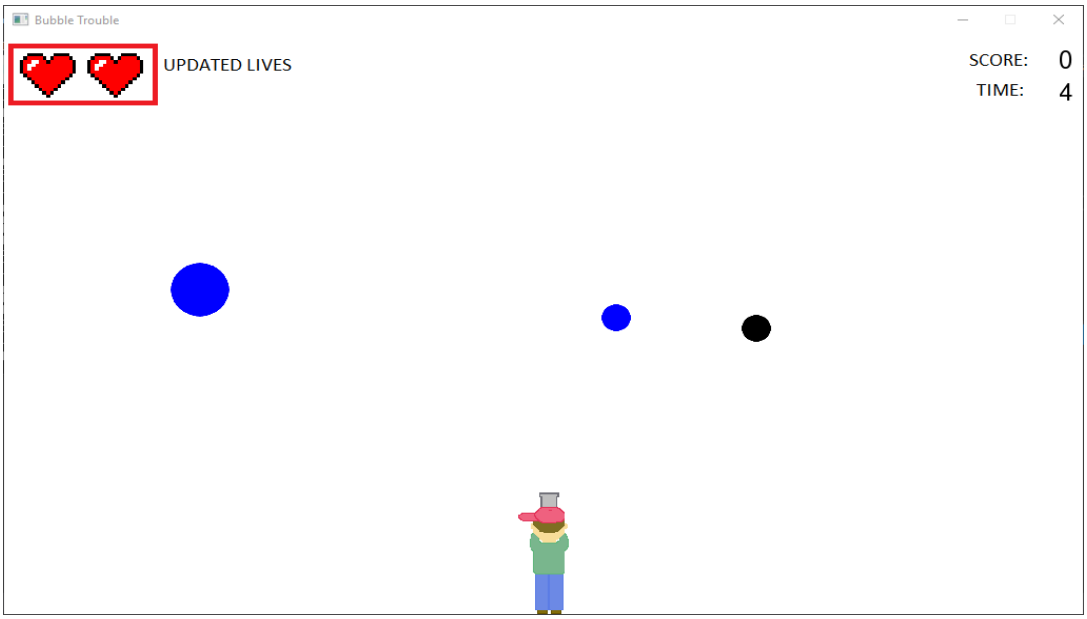

Figure-4

Figure 4 shows that when a bubble has collided with the player, one life is lost and is updated in the lives section as shown in **figure 2.**

Use case diagram

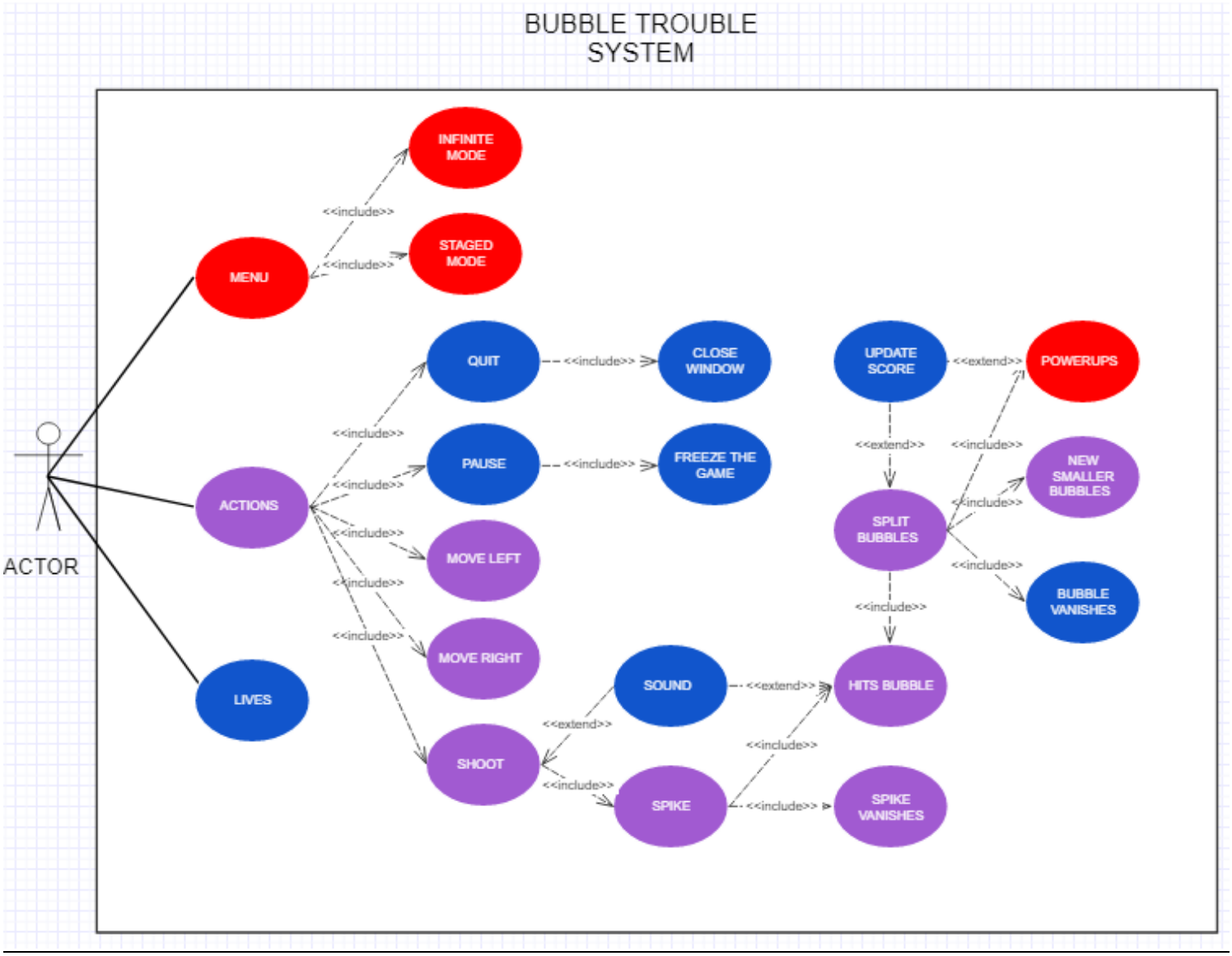

Figure-5 Use case diagram of the game

Use case diagram has three main use cases and rest are either extended(“extend”) from a use case or part of the same use case (“Include”).

The main use cases that an actor/player can see in the Bubble Trouble system is

- Actions
- Lives

- Menu

A player ‘s action includes

- Moving left
- Moving right
- Shoot the spike
    - Hit’s a bubble
        - New bubbles are created
        - Explosion sound is heard with a small explosion image around the newly formed bubbles
        - Score is updated
- Pausing the game &
- Exiting from the game

When the game is started, a menu must be seen, and a player might select if he wants to play an infinite game mode or staged mode. Also, certain power ups such as more lives, increasing time etc. would be made available to a player to collect it. These use cases would be explained in detail in cycle 3.

Inheritance Diagrams

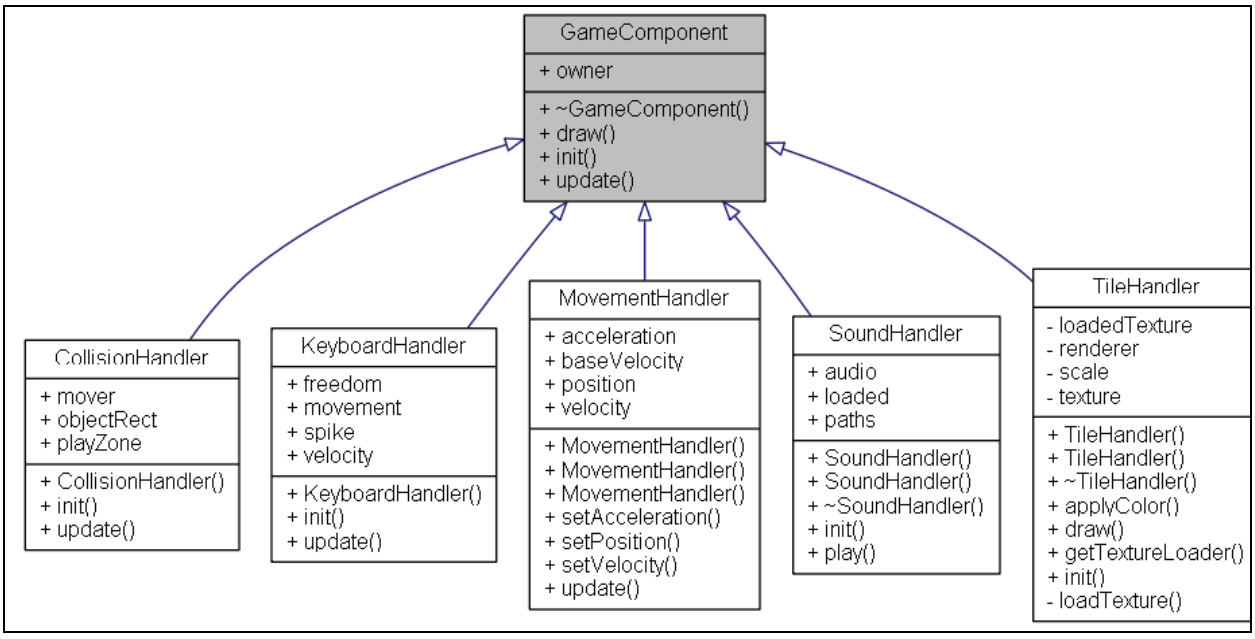

Figure-6 Inheritance diagram of GameComponent class

The GameComponent class is the base class and each “Xhandler” class inherits from it with its own functionality.

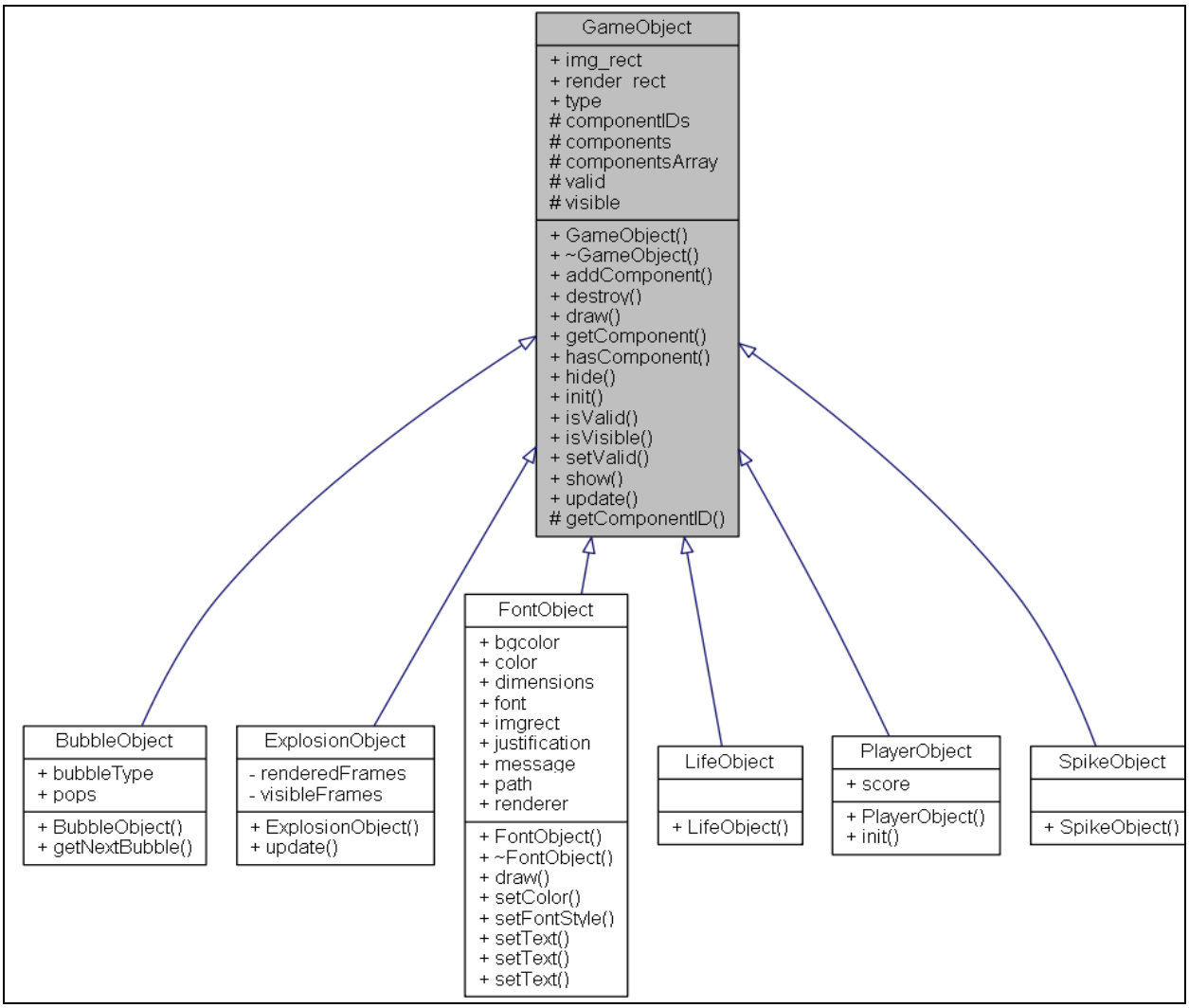

Figure-7 Inheritance diagram of GameObject class

The GaneObject class has a few inherited objects for each type of object currently in the game.

Class Diagram

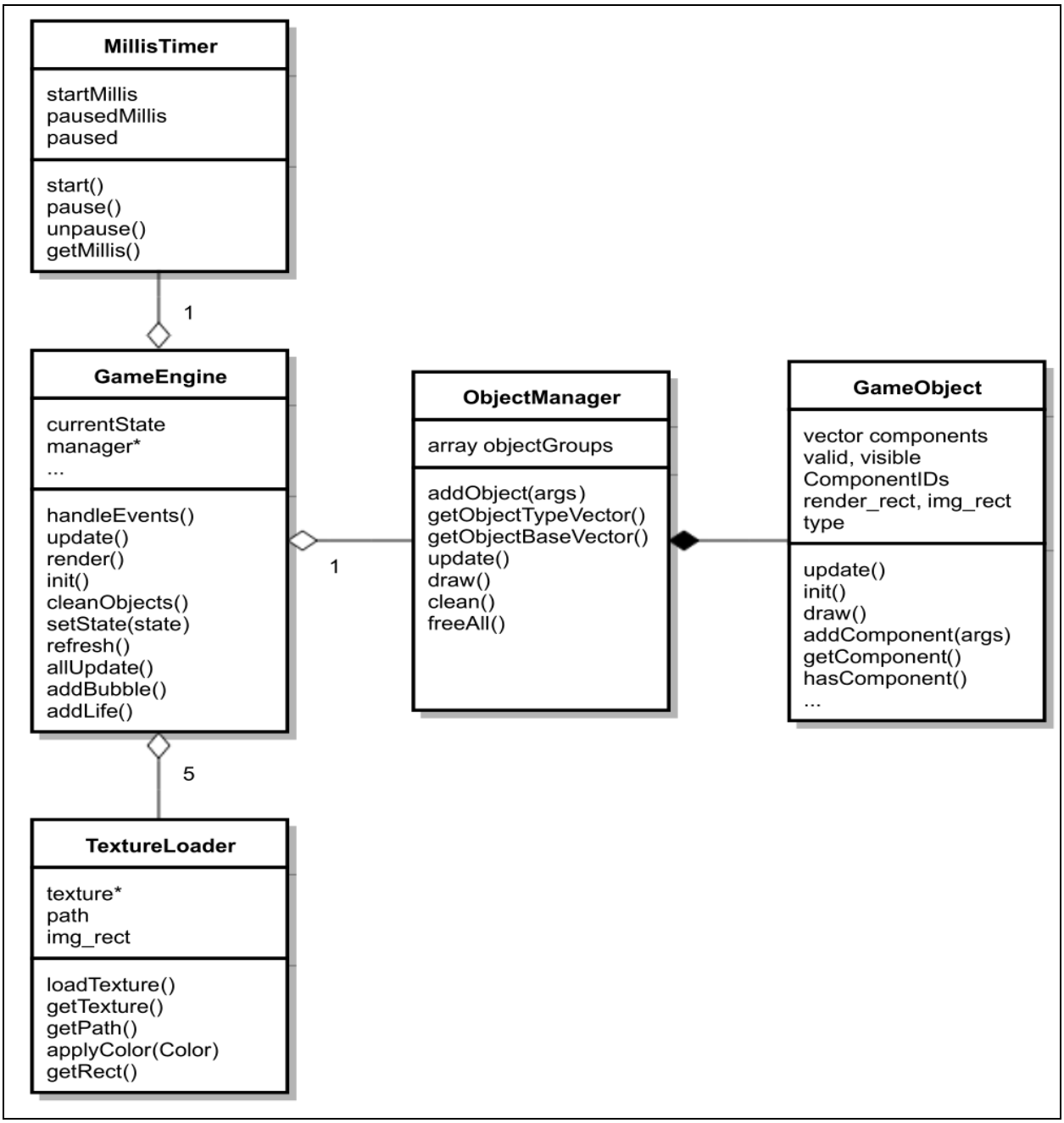

Figure-8 Class diagram of the important classes being used in the game

Omitted is the inheritance of the game object types as well as the components' connection to the GameObject class.

Recap of division of work for cycle 2

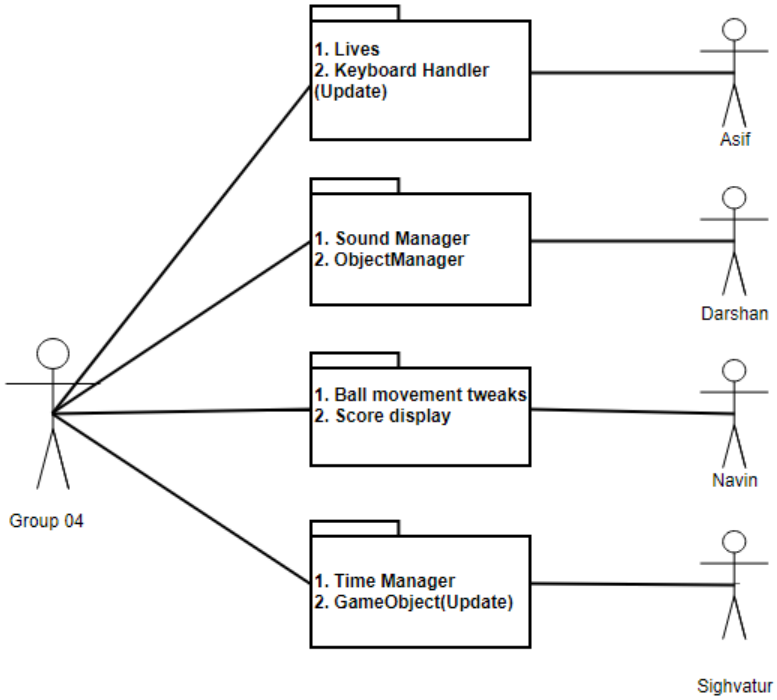

Figure-9 Work Division for cycle 2

Division of work for cycle 3

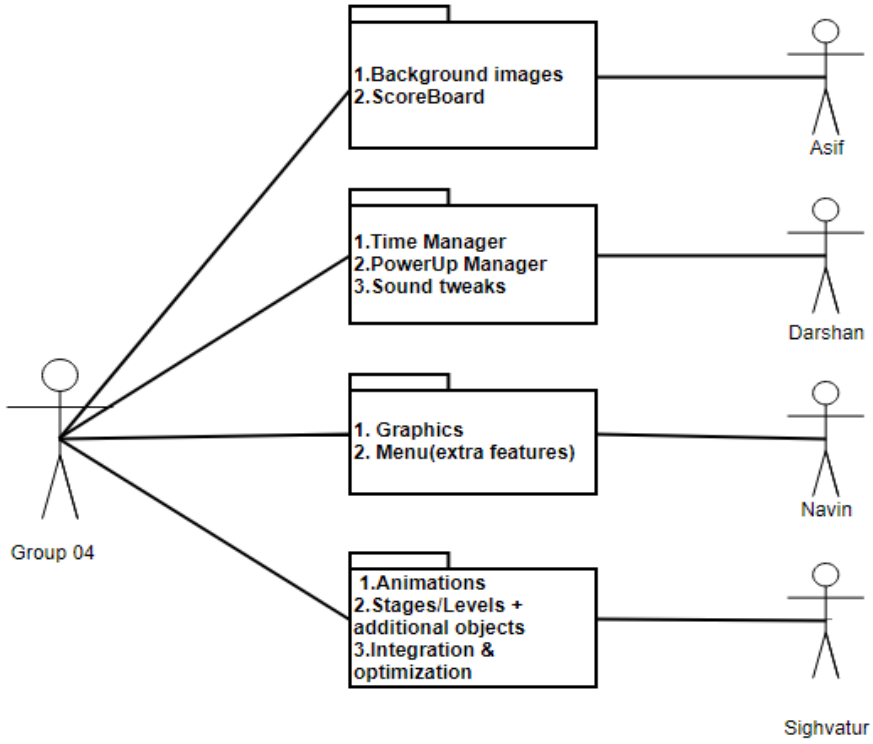

Figure-10 Work Division for cycle 3

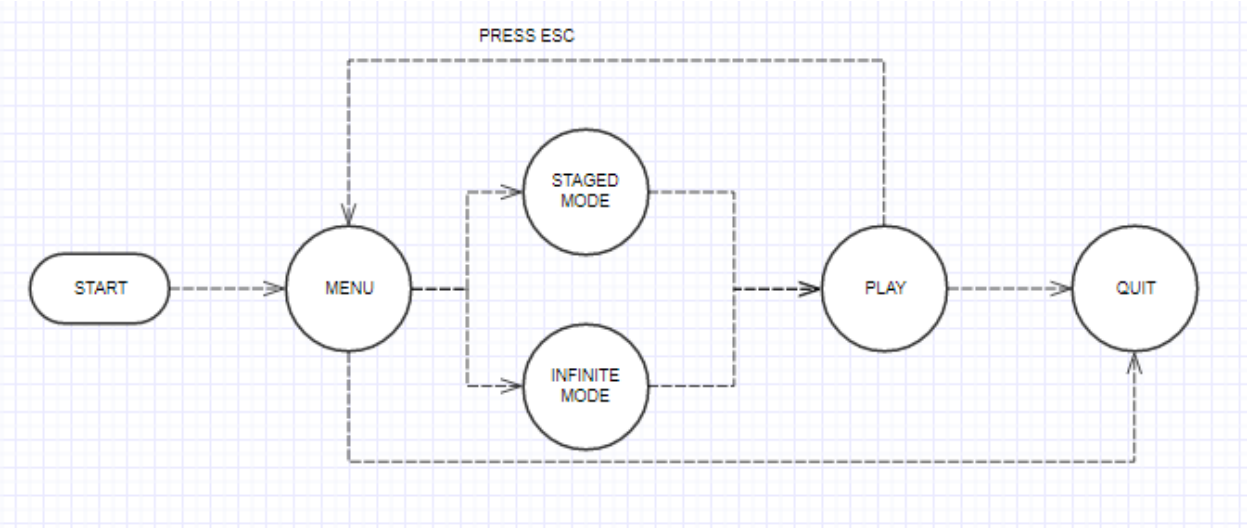

Figure-11 Flow Diagram for Cycle 3

Flow diagram of the gameplay is shown above. The game starts, and the menu is displayed with options such as Staged mode and Infinite mode. The player can choose either of them and enters the playing area with three initial lives. If an ESC is pressed during gameplay menu will be shown which will have options such as resume and quit. Player can 'X' out at any time during the gameplay.