Phase 2 Report | CMPT 276 Project Group 5

Members: Kiran Khubbar, Amar Mondair, Prabhjot Singh, Karanveer, Singh

Approach:

The first step in creating our game was learning the fundamentals of 2D game development. We did this by researching and watching YouTube tutorials to understand how different parts of a game work together. Some examples are: movement, collision detection, map rendering, and sprite animation. Once we understood the basic structure, we used that knowledge as a foundation to design and build our own custom game. While the tutorials helped us grasp the core mechanics, most of our systems, such as enemy AI, object interaction, and map design, were created and adapted specifically for our project. We also used external tools to make the game more unique and polished—Piskel.com for designing custom pixel art sprites and BeepBox for creating original sound effects and background music.

Use Case Changes:

During the development, several adjustments were made from use cases. The original use cases assumed a strict room-based movement system, fixed interactions with keys, punishments, and ghosts.

In the implementation, we transitioned to a tile-based movement system which required updates to the player and monster movement logic and collision handling (UC2). Player and monster movement needed direction logic and checking each tile for collision.

Phase 1 design assumed multiple keys placed manually in the map and was kept the same (UC3).

The punishment mechanism was also simplified and transitioned into a collision damage system (UC4).

The biggest change was ghost/monster behaviour (UC5). Phase 1 described ghost simply checking if they are in the same room. In implementations, this was changed to an AI-type system where the monster runs towards the player and damages the player. This change improves difficulty and engagement beyond what the early use cases assumed. This was also replaced with a more dynamic system in one of the levels, where the monster is tracking the player by comparing worldX and worldY positional coordinates.

Finally, the boss fight and win conditions (UC6) were not changed and match the actual scope of design. The player will still need 2 keys to enter the basement and will need 3 money to win the game. Overall, the modifications align with the design, and changes were necessary for a more engaging gameplay.

UML Diagram Changes :

In the initial design, our UML diagram outlined a detailed object-oriented structure that included many subclasses such as MovingEnemy, Punishment, RegularReward, BonusReward, and specialized bosses like MichaelMyers, EvilSanta, and FinalBoss. It also included supporting classes like Board, Barriers, and GameState to manage the game's logic and state.

As the project progressed, several modifications were made to simplify and optimize the implementation. The largest change was consolidating multiple enemy and reward subclasses into generalized systems. For example, instead of separate enemy subclasses like MovingEnemy or Punishment, we created a single Entity superclass with flexible attributes and behaviors. Specific enemies, such as the boss, extend Entity and define unique behavior (e.g., firing projectiles) through the setAction() method. This reduced code duplication and made it easier to manage all entities using shared logic.

Similarly, instead of multiple Reward types, we implemented different object classes (e.g., Obj_Heart, Obj_Money, Obj_Chest_Key) that all extend from Entity. This unified approach simplified the collision and pickup systems, allowing a single collision checker to handle all object interactions.

We also replaced the Board and Tile system with a dedicated TileManager class, which handles map loading, rendering, and collision detection. This design better suited a tile-based game engine and allowed for easy map expansion using text files.

Finally, rather than managing the game through a GameState class, we used state variables directly in GamePanel (e.g., playState, pauseState, titleState) to control transitions between game screens. This made the structure more lightweight and easier to debug.

Overall, these adjustments were made to make the code more modular, maintainable, and compatible with real-time game loops. The simplified class hierarchy allowed smoother integration and better performance while maintaining all planned gameplay functionality.

Responsibilities:

Kiran Khubbar

- Pathfinding for enemies, the final level (basement) of the game, player implementation, player attack with mele and projectile, player inventory, important items handling (keys for doors, money for score), pause screen with quit option and key controls displayed, game over screen, game win screen (if conditions are met), dialogue with NPC screen, NPC implementation, key handling, implementation of game structure files (GamePanel, AssetSetter, CollissionChecker, UI, Sound, TileManager, Entity), implementation of final boss, handling of damage and lives of both player and final boss.

Amar Mondair

- Lobby level design and implementation, level handling through teleportation tile to different maps, NPC implementation in lobby, code Integration (adding group members' code to lessen conflicts because of tight deadline), committing code, pushing code, and handling any errors that arose from adding code such as, map dimension errors, Game Thread errors, player not moving, maps not loading, etc.

Karanveer Singh

- Enemy implementation (Michael Myers), object system (key), map for Myers level, tile-based collision and sprite animations for standing and running, key system with random hidden key placement under doors, hotel upstairs hallway design with doors as rooms and collision tiles for difficulty, player damage when Myers catches them, implementation of MyersEnemy, Key, and parts of GamePanel, TileManager, AssetSetter, and CollisionChecker for Myers level.

Prabhjot Singh

- Implemented Evil Santa Class that extends entity created object classes for gifts and snowballs Created a Christmas themed map and designed appropriate tiles and music for the level

External Libraries Used:

1. javax.swing
   Used for the main game window and screen drawing.
   Classes like JPanel and JFrame provide the framework for rendering the game and managing the GUI (e.g., the GamePanel).
2. java.awt
   Provides the graphics and drawing tools (e.g., Graphics, Graphics2D, Color, Rectangle).
   It's essential for rendering sprites, hitboxes, and handling 2D visuals.
3. java.awt.image.BufferedImage
   Used to store and manipulate images such as tiles, entities, and UI icons.
   Every sprite or object graphic in the game is handled as a BufferedImage.
4. javax.imageio.ImageIO
   Handles loading image files from resources (e.g., PNGs for tiles, characters, and objects).
   It's simple and efficient for sprite management.
5. java.awt.event (specifically KeyEvent, KeyListener)
   Used for keyboard input detection.
   This allows the player to move the character and interact with the game world.
6. javax.sound.sampled
   Used for music and sound effects (AudioInputStream, Clip, AudioSystem).
   This enables background music and sounds for actions like picking up items or opening chests.
7. java.io
   Used for file and resource input, such as reading map files (BufferedReader, InputStreamReader, InputStream).
   This allows the game to load level data from .txt files.
8. java.net.URL
   Used to locate and load sound or image resources bundled within the project's directory structure.
9. java.util (e.g., ArrayList, Collections, Comparator, Random)
   Provides data structures and utilities for managing entities and randomizing behaviors (like monster movement or item drops).
   Also used to sort entities for rendering order.

Challenges:

The biggest challenge we faced during this phase was managing the conflicts that arose within our codebase. During development, each team member had their own smaller

version of the GamePanel class used to draw and test the game screen. While this made it easier for everyone to work independently at first, it caused major issues later when we tried to merge all the versions together. We often ran into merge conflicts and errors caused by slightly different versions of the same code. Because we were working under a tight deadline to get the main game structure functioning, we decided to minimize these conflicts by appointing one person to handle integration. This person was responsible for copying, pasting, adding, committing, and pushing the final versions of code from each teammate. This approach helped streamline our workflow, reduce errors, and make it much easier to test and implement new features efficiently before the deadline.