Phase 4 Report

The Game

Our game revolves around the Grim Reaper who navigates through a maze to find souls to bring to the afterlife. But he must watch out for the moving angels trying to catch him, who won't hesitate to execute him immediately, as well as the evil red souls. Bonus gold souls will appear from time to time, which Grim may collect.


Player

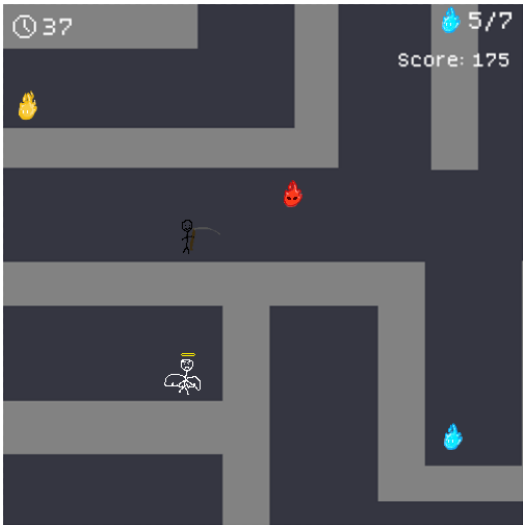
Enemy


Regular Reward


Bonus Reward


Punishment

In our initial mockup, we planned to make navigating through the maze more challenging by not showing the entire game board and have a camera track the main character in a square-shaped window.



We ended up not implementing camera tracking while coding the board, as it was an extra addition we didn’t need and we merely wanted the board to work at the time due to time constraints. The small board was also too difficult for the player to navigate through, so the window size had to be expanded. We should have researched and planned more thoroughly during the design phase to avoid extra changes. This has taught us how important the design phase is. The implementation phase should not have been where we figured out how most components would work.

The enemy movement also has proved to be an unexpectedly challenging task. Probably because our implementation of the movement was based on class Point rather than on the custom Cell and Board classes. Point class represents a pixel, which proved common pathfinding algorithms such as A* and Iterative deepening A* to be too expensive for our computers to compute. Using the abstract cells for movement should have been considered earlier for the pathfinding algorithms to function on time.

As for user input, we did not have a concrete idea on implementing it in our design as we were not familiar with this topic. Between KeyListener and Keybinding, we decided to go with KeyListener as it is easier to implement codewise. The consequence is that we have to consider focusability. But it wasn't a big issue as the only component that takes user input is the JFrame and all we need to do is to make JFrame the only focusable component.

On the topic of game difficulty, we had to repeatedly tweak and test the game to find a comfortable balance between ease and challenge. During the implementation and testing phases, we changed some spawn styles from hardcoded to randomized, lowered angel speed, altered the size of pathways, and changed the layout of walls. Balance is a very subjective topic and can be easily affected by small changes in implementation, so these edits were inevitable. Unlike some elements like board size and camera tracking which should have been researched more thoroughly in the design phase, these small details were more difficult to predict and required continuous improvements. This experience emphasizes the importance of ongoing testing and re-challenging of prior design during implementation, because you don't know exactly how the game will feel until it is being implemented.

Our Game object was supposed to contain many fields and methods related to each round of the game. For example, it checked if the round was over, handled collisions between entities, and contained a Controller object which initiates and moves entities. Upon further evaluation, we decided to move these elements to GameState instead. That way, Game could focus on overarching app functionality like state switching, while round-related code is kept safe in GameState. Game could also call the current state's tick() method without worrying about the contents of tick(). This demonstrated the usefulness of separation of concerns by simplifying Game code and reducing mental overload during implementation and testing.

We began with three states: MenuState, GameState, and EndState. During implementation, we split EndState into WinEndState and Lose EndState to better reflect the possible game results. This could also reduce the gulf of evaluation by allowing users to quickly grasp the state of the game (their win or their loss) and formulate their next steps (how to continue winning or stop losing). We also added an InstructionsState for the sake of the player, so that they were able to learn or review the rules of the game before starting a round. The driving force between that decision was to improve the user experience. Adding instructions is a form of help and documentation, which can reduce learning time and improve user feeling of control.

We expected to use the Singleton design pattern, but were somewhat surprised at how much we used it and how handy it became. Using singletons to represent states allowed us to maintain important information throughout ticks, ensured that all objects could access it easily, and let us avoid needless destruction and recreation of states. We also made the Controller a singleton so that we didn't create multiple sets of characters and items when one would suffice. Singletons also contributed to our mental image of the code, allowing us to understand them more concretely as individual building blocks that could fit together, rather than being overwhelmed by potentially infinite numbers of instances.

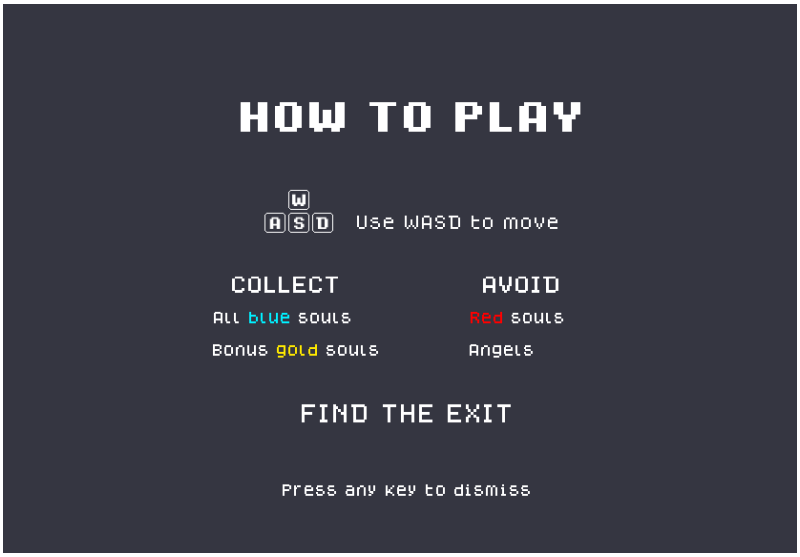
Phase 4 Report

Overall, we learned the importance of preparation in the early project stages. By carefully brainstorming aspects of our game instead of jumping in prematurely, we were able to create useful resources like theme descriptions, class diagrams, and use cases. These resources helped us divide the work among team members and served as guidelines to refer to during implementation. Whenever we wanted to add or alter code, we could explicitly identify the change by comparing our proposed changes to the original diagram, and easily discuss them with the team. The concrete visuals also let us identify opportunities for creational design patterns like state and singletons. Finally, these resources ensured that coders were on the same page about our expectations for the game. These benefits lasted in every phase, from implementation to testing to wrap-up reports. Meanwhile, when we faced problems like too-high expectations of camera tracking or improper separation of concerns, this was usually due to a lack of foresight while creating the initial design. Ultimately this experience taught us that effective software engineering is not just about proper syntax or software use. Rather, it should incorporate a design process with careful research of requirements, application of design principles, thorough documentation, and collaboration between team members.

Tutorial

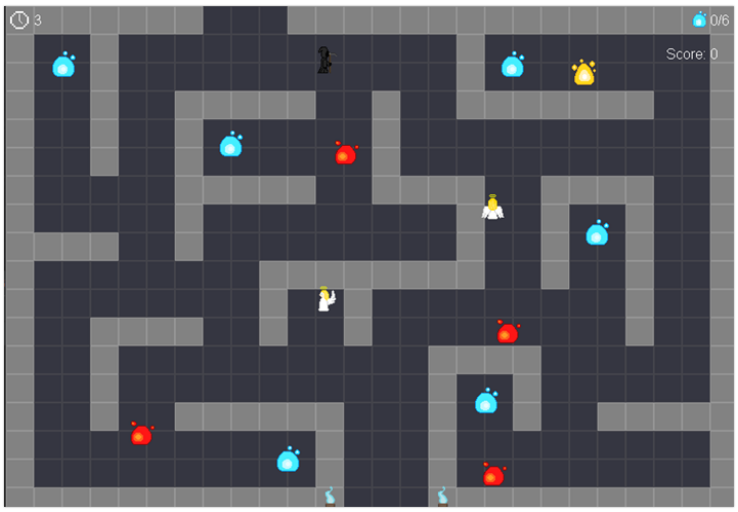
When the game is launched, the user will encounter the menu screen, in which they can press any key to proceed to the instructions. At every stage of the game, the player can easily spot their next steps due to the high contrast between the white text and dark grey background, which also helps form a recognizable identity to the design.

The instructions show that the Grim Reaper is controlled using the WASD keys, and that they must collect all blue souls, avoid angels and red souls, collect bonus gold souls and find the exit. The game begins once the user presses any key.



Once the game starts, the player will spawn at the top of the maze and must get to all the blue souls and make it to the exit (indicated by the blue torches) without dying. Colliding into a blue

soul rewards the player with 50 points, gold souls reward 100 points, red souls deduct 75 points and an Angel will immediately end the game. Gold souls and punishments spawn randomly to add an element of chaos and challenge to the game.



Once the player either wins or loses, the win/lose end screens will appear, which displays the player's final score, blue and gold soul count and time elapsed. The player can highlight either the Play Again or Menu buttons using the W and S keys as up and down arrows. Pressing enter will finalize their selection.

The user can close the game by clicking the X button in the top-right corner of the window.