Implementation

Repository can be found at: https://github.com/sepr-smew/mallard.git

AI (Change IM10)

When we initially received the project, we found that the AI tended to give up quickly, reducing how challenging the game was. To rectify this, the AI was heavily modified. The modified AI performs a greedy best first search to find a path to the player. We chose to use this algorithm because it provides a good compromise between complexity and efficiency, with alternatives such as A* requiring more complexity for testing. Once locating the Duck, the AI simplifies its path by raycasting from its current location to each of the nodes in its proposed path, until it reaches the last node. Using this node, intervening nodes can be eliminated, producing smooth movement.

Minimap (Change IM4)

As part of this assessment, we implemented a minimap in the bottom left hand corner of the window. This minimap makes use of a new viewport and camera, to render a wider view of the map from the same position of the game camera, essentially giving a smaller, more zoomed out, version of what the player can see in the main window. To make the minimap clearer, a different texture is used when rendering the duck. Enemies are also rendered differently, using red dots drawn with a ShapeRenderer.

Box2D (Change IM2)

Previously, the physics system was held in the Round class. This meant that the Round class contained a lot of lines of code, making it difficult to read. Implementing Box2D, a physics library provided by libGDX, we removed a large amount of logic from Round. Box2D also had the added benefit of dramatically increasing the efficiency of the game, effectively doubling the frame rate of lower powered computers (high power computers ran at a capped framerate of 60FPS regardless).

Water (Change IM5)

Water is implemented as a layer in the tiled map that is converted to a box2D body when the map is loaded. Box2D is used to translate this to changes in movement type, by allowing us to easily create a listener that switches the player to a swimming state on entry, and a walking state on exit.

Level Selection and Persistence (Change IM7)

The requirements (G9) state that the player state should persist between levels. Since the only object that persists throughout the game is the main DuckGame class, it stores data for the current session in a single inner class. This allows a game to be easily reset without having to restart the instance.

Requirements Met	Description
G1, G2, G6	Our product contains 8 rounds (G1), each with it's own location and objective (G2). Rounds include: 1. Alcuin -Find the flag 2. James - Find the flag 3. Halifax - Kill all enemies 4. Quiet Place - Survival 5. Heslington West Lake - Survival 6. Library - Survival 7. Heslington East - Find the flag 8. Computer Science - Boss fight It can be seen that all locations exist within the University of York (G6)
G3	Each level is designed to be played within a 5 minute period (G3). Only objective is present per round, and the objectives themselves are clear and simple.
G4, I2	A point system exists (G4), with the score clearly visible in the top left hand corner of the window (I2). The points are held within the Session class.
G5	A health system exists (G5) with the current health clearly visible in the bottom right hand corner of the screen, shown using hearts. The current health is held within the Session Class
G7, G8	Within the game, two types of obstacle are implemented (G7): 1. Static obstacles, such as bushes, trees, buildings and rocks. These are implemented using tiled map editor, with obstacles stored in a separate layer to non collidable objects. 2. Enemies (G8), such as ranged geese, melee geese and the mecha-boss. These are implemented using the Mob class. These obstacles are placed in the level randomly/pseudo-randomly: 1. In the case of static obstacles, a pre-defined configuration is selected at random. 2. Enemies are placed randomly (G7)
G9, G10	The player is able to obtain resources, including health and weapons (G10). The weapons have their own properties and can be used to damage enemies.
G11	The player can obtain four powers (G11), invulnerability, increased rate of fire, shield and score boost. These are implemented using the Pickup class.
G12	When a level is failed, a fail screen appears (G12). In order to fail a level, the player must run out of health. This is implemented using the LoseScreen class.
G13	When the final level is completed, the WinScreen class is used to show the player that they have finished the game (G13).
I1	The game uses a consistent art style throughout, making use of pixel art assets (I1).
I3	The background for the game was created using the tiled map editor, using 2D assets. This combined with the pixel art sprites, gives the game a pseudo 3D look (I3)
I4	A minimap is present at all times during Gameplay (I4). The minimap contains the

	location of the player as well as nearby enemies. This is implemented using the Round class.
C1	The player can navigate the game using the WASD keys. Implemented using the Player class and Box2D worlds.
C2	The game implements three types of movement (C2), waddling, swimming, and flying. 1. Waddling is a relatively slow, implemented using the Player Class (C3). 2. Swimming make use of movement logic handled by Box2D. The boundary between water and ground is defined using separate layers within the tiled map (C4). 3. Flying is the fastest method of movement, but has a limited (but regenerating) time of use (C5).