Ethan Miller

Design Document for:

DREAD OF KNIGHT

Written by Ethan Miller

Version # 1.10

December 4, 2023

Game Overview

This is **DREAD OF KNIGHT**, a 2D top-down, horror maze game where the **player** plays as a knight trying to escape a haunted castle.

Mechanics/Scripting

Player Movement Mechanics

The script defines a public float variable called moveSpeed, which determines the movement speed of the player. It also declares a private variable called rb, which is of type Rigidbody2D, and will be used to store the Rigidbody component of the player. In the Start function, the script gets the Rigidbody component of the player and stores it in the rb variable. In the Update function, the script reads the horizontal and vertical input axis using Input.GetAxis("Horizontal") and Input.GetAxis("Vertical"), and stores the values in the moveHorizontal and moveVertical variables. Then, the script checks if the absolute value of moveHorizontal is greater than the absolute value of moveVertical. If so, it sets moveVertical to 0, and vice versa. This is to ensure that the player moves in the correct direction when both horizontal and vertical input are given. After that, the script sets the velocity of the player using the Rigidbody component, by multiplying the input axis values with the moveSpeed variable. It also sets the appropriate values for the Horizontal, Vertical and Speed parameters of the animator, to control the animation of the player character.

Enemy Movement Mechanics

The script sets some public variables to configure the enemy's movement behavior, such as the speed of the enemy, the distance the enemy needs to be from the player to start dashing, and the duration of the enemy's dash. The script then sets up some private variables, such as a reference to the player object, and an enum to represent the different states of the enemy's movement, such as moving towards the player, pausing, and dashing. The script has several functions that define the enemy's behavior in each state. The MoveTowardsPlayer function moves the enemy towards the player until the enemy is close enough to start dashing. The Pause function makes the enemy pause for a set amount of time before dashing towards the player. The Dash function makes the enemy dash towards the player for a set duration. The EndDash function ends the dash by setting a flag indicating that the enemy is no longer dashing. The script also includes code for rotating the enemy to face the player and changing the sprite of the enemy to reflect whether it is dashing or idle. Overall, the script is used to control the movement of an enemy in a game by making it move towards the player, pause, and dash towards the player.

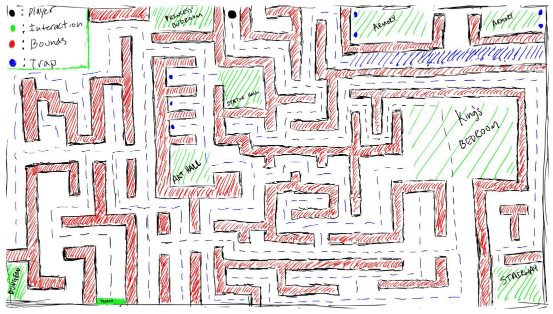
Camera & Light Mechanics

The first script, called CameraFollow, is a script that is attached to a camera in a Unity scene. The purpose of this script is to make the camera follow the player as they move around the scene. It does this by first creating a reference to the player's transform object, which is used to get the player's position. Then, in the FixedUpdate() function (which runs every fixed frame update), it sets the target position for the camera to be the player's position plus a fixed offset on the z-axis. The script then uses the SmoothDamp() function to smoothly move the camera towards the target position. The velocity variable is used to keep track of the camera's current velocity, and is passed by reference to the SmoothDamp() function so that it can be updated by the function. The second script, called LightFollow, is similar to the first script, but it is used to make a light in the scene follow the player instead of a camera.

Maps

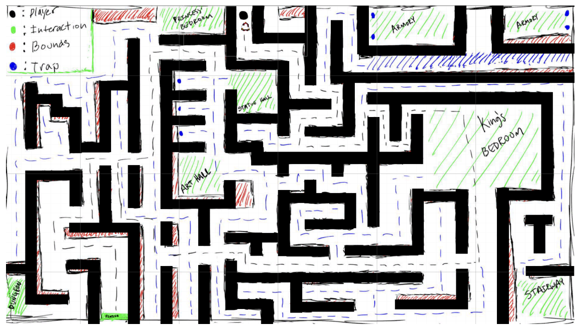
The Castle

Version 0.1



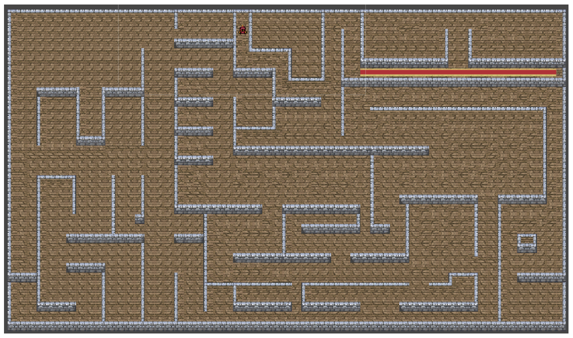
This is an alpha map of Level One. This was sketched and shows the potential pathways a **player** can take. **Black** dotted lines represent the correct paths a **player** can take to "win" the level. **Blue** dotted lines represent the wrong paths a **player** can take to kill them or "stall" the level. The spaces colored in **Green** show the places in the level that a **player** can "interact" with. These areas may transport the **player** to different scenes or could have items that a **player** can interact with and learn more about.

Version 0.2



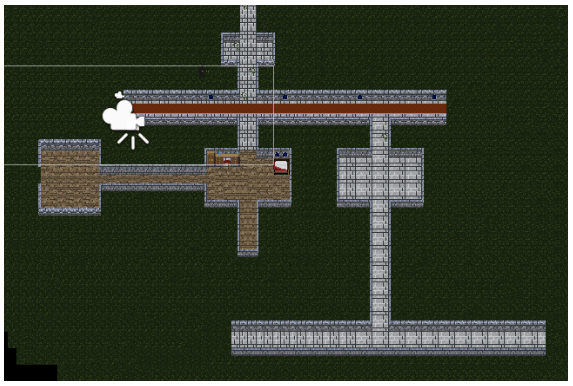
This is an alpha map imported into Unity. Using game objects as walls, the maze is pathed out correctly. The **player** can not pass these walls.

Version 1.0



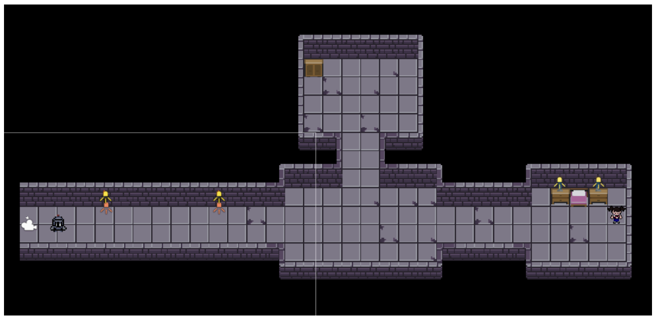
This map was created using Unity's tilemap. Since the environment is supposed to be a castle, I made hardwood floors and cobblestone walls.

Version 1.1



This map was created using Unity's tilemap. I thought covering the entire floor in hardwood looked weird, so I made most of the castle have marble floors, opting for only the hardwood to be used in certain rooms.

The Dungeon



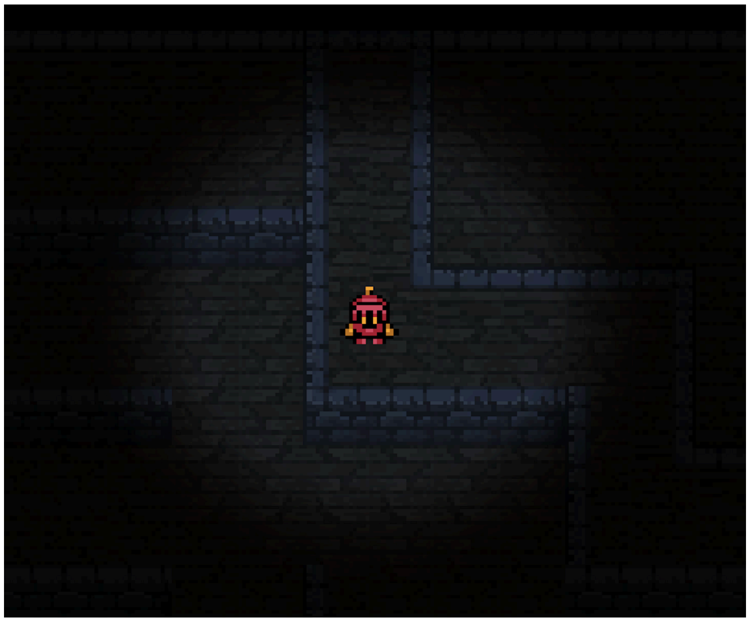
This map was created using Unity's tilemap. The map is small, but the point is to talk to the **dungeon girl**, who is located at the far right. Interacting with her will give the player the **key**.

Player POV (Camera)

Version 0.1

The camera will be close-up and follow the player. In further iterations of the game, the “view” that the **player** sees will be a small spotlight around them.

Version 1.0



In this camera view, there is a small circle of light around the **player**. This adds to the horrifying feel of the game, and makes the maze component much harder to operate.

User Interface

The **player** will move using standard “WASD”. The **player** does not move diagonally; they are forced to strictly adhere to the Y and X axis. This decision was made to (a) limit the **player’s** movement capabilities while dodging enemies and (b) mimic the control scheme of old top-down Nintendo games. W and S allow for **player** movement up and down the Y axis, whereas A and D allow for **player** movement along the X axis.

The speed of the **player** is a slow walking speed. In a meta sense, if a knight truly was creeping around a haunted mansion, they would likely tread carefully due to unknown threats.

The **player** can press “I” to open an inventory. The inventory will show the **player** whether or not they have the **key**.

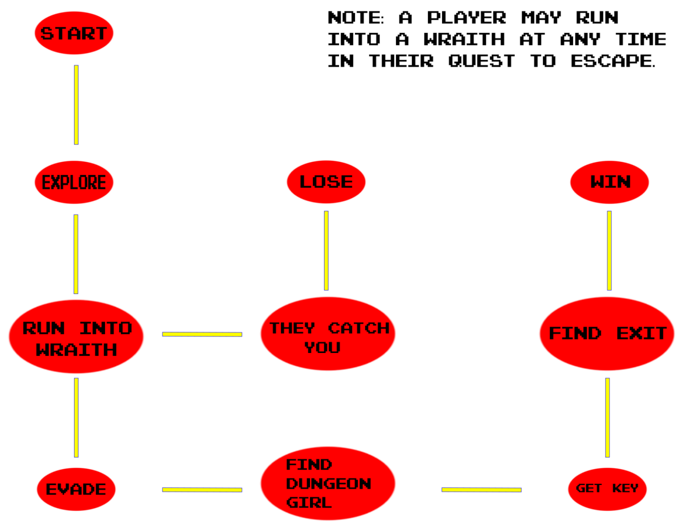
The **player** can press “ESCAPE” to bring up a menu where they can exit or restart. This menu also appears when the **player** is caught.

Rules & Win Conditions

Considering this is an electronic game, the rules are automated. However, this does not mean that laying out the rules is futile.

1. The **player** may not step out of the game’s boundaries.
2. The **player** may not be touched by any of the wraiths.
3. The **player** must retrieve the **key**.
4. The **player** must reach the **exit**.

Below is a flowchart, showing what gameplay should look like.



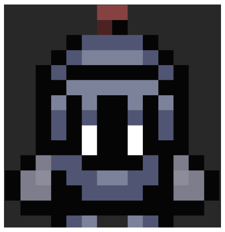
Feature Set

1. 16-bit graphics
2. 16-bit color scheme
3. Finite maze

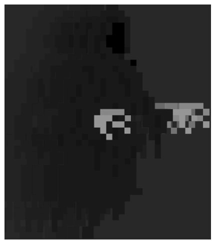
4. Top-down Camera
5. 2D Graphics
6. Inventory System

Characters

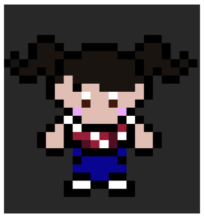
There are two main characters and one interactable NPC. The most important character is the **player**. The **player** is being attacked by **wraiths**.The **player** can interact with the **dungeon girl** by traveling down the stairwell; the **dungeon girl** will give the **player** a key, which they can use to exit the castle.



Above is the **knight**. This is who the **player** will play as.



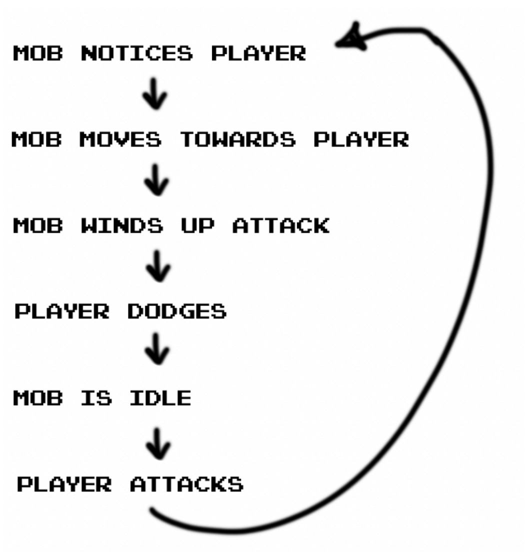
Above is the **wraith**. This is the story's antagonist.



Above is the **dungeon girl**. The player will have to find the **dungeon girl** in the stairwell to retrieve the **key**.

Combat Mechanics

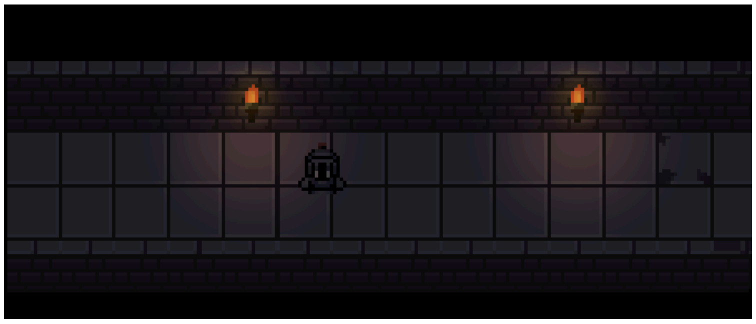
The combat will take place in rooms. There will be a plethora of different enemies such as orcs, goblins, and ghosts. Each mob will have their own "style" of fighting. This can be but is not limited to: speed of attack, speed of movement, and intelligence. Considering this is a top-down game, the player will be able to "swing" their sword in the direction they are facing. If there is a mob within a tile of the player, the player will deal damage.



Above is a flow chart for how combat will progress.

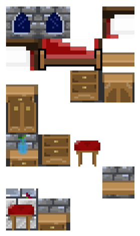
Lighting & FX

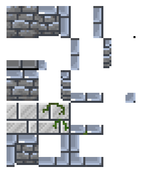
The game will use Unity's Universal Render Pipeline(URP) to render realistic lighting in a 2D environment.

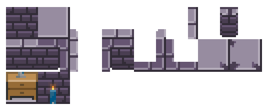


Above is an example of a torch with lighting properties.

Art & Miscellaneous


Above is the bedroom tileset.


Above is the castle tileset.


Above is the dungeon tileset.

Above is the bed the player will interact with.

Musical Score & Sound FX

The music for this game will be mashed to 32-bit audio. There will be a strong use of diminished and augmented chords.

When the **player** moves, there will be a creaky wood sound effect to accompany their steps.

When a **wraith** is close to a **player**, there will be a low groan to warn the player of impending doom.

Playtesting

There are several elements of this game that can be playtested.

The Maze

The maze must be difficult, but not impossible. There must be multiple pathways to lose and multiple pathways to win. This was playtested by sketching out different mazes and finding which one was the hardest, but also most satisfying to complete.

Wraiths

The **wraiths** must pose a threat, but not one that will overwhelm the **player**. In my scripting section, I detailed the coding that went into creating the **wraiths**. I can manipulate the **wraith's** speed, dash, and dash distance. This mechanic is still being playtested. I will ask peers to playtest my game and give me feedback.

The Combat

The **player** will click MOUSE1 to "attack" in whatever direction they are facing. This will be playtested by my peers and I. If there are any mechanical problems, they will be fixed in future iterations.

Player Movement

The **player** must be able to move quickly, but I wanted them to feel limited in their options. To keep the dramatic elements of the game sound, I decided to make the **player** move slightly slower than one normally would in a top-down 2D game. This is because the **player** is scared and ready to fight. The **player** can not move in any diagonal direction. This is to emulate the feel of old NES and Gameboy games.

Budget/Development

1. Game Design & Documentation
   - Define game mechanics and scripting logic: 10 hours
   - Design player movement mechanics: 5 hours
   - Design enemy movement mechanics: 8 hours
   - Design camera and light mechanics: 6 hours
   - Create map layouts and iterations: 12 hours
   - Design player point of view (camera): 4 hours
   - Design user interface and controls: 6 hours
   - Define combat mechanics: 8 hours
   - Determine rules and win conditions: 4 hours
   - Create flowchart for gameplay: 4 hours
   - Total: 67 hours

2. Art & Graphics
   - Create 16-bit graphics and color scheme: 20 hours
   - Design tilesets for castle and dungeon: 12 hours
   - Create character sprites for player, wraiths, and dungeon girl: 16 hours
   - Design and create additional art assets (e.g., bed): 8 hours

- Total: 56 hours

3. Audio & Sound FX
    - Compose musical score in 32-bit audio: 10 hours
    - Create sound effects (e.g., footsteps, wraith groans): 8 hours
    - Total: 18 hours

4. Programming & Development
    - Implement player movement mechanics: 10 hours
    - Implement enemy movement mechanics: 12 hours
    - Develop camera and light mechanics: 8 hours
    - Implement map generation and tilesets: 14 hours
    - Implement player point of view (camera): 6 hours
    - Develop user interface and controls: 10 hours
    - Implement combat mechanics: 16 hours
    - Total: 76 hours

5. Playtesting & Iteration
    - Test and refine maze design: 10 hours
    - Playtest and adjust wraith behavior: 8 hours
    - Playtest combat mechanics: 6 hours
    - Refine player movement and controls: 4 hours
    - Total: 28 hours

Considering an estimated hourly rate of $50, the total estimated budget for this project would be:

Total Hours: 245 // Hourly Rate: $50

Estimated Budget: 245 Hours*$50/Hour = $12,250