User Documentation

Gameplay Summary:

2-D rogue-like dungeon crawler with a gambling-based system.

A two-dimensional top-down view of a character whose objective is to complete unique levels. To win a level, the player needs to defeat a certain number of enemies or survive for a set amount of time. Upon completion, one of the room's four doors opens, leading the player to the next room. Upon entering the new room, the player is presented with the characteristics of the next level, allowing them to make an informed betting decision. A shop opens, allowing the player to wager money, health, and other game aspects, like boosting enemy damage, to win more money at the end of the level. Each level gets progressively more difficult, and if the player does not bet, their score will be lower than it otherwise could be. When a player loses all of their health, they lose the run. A leaderboard prompts the player to enter a username and displays their score. Win money to get the highest score!

Installation:

Play Raging Gambler on Itch.io in the browser window!

https://miguelchaveznava.itch.io/raging-gambler

Gameplay:

Main Menu

1. Start Run: Directs player to normal arcade mode.
2. Tutorial: Directs the player to a sandbox-style tutorial mode where the score is not saved. Tutorial mode allows the player to develop strategies and get a feel for how the game is played inside a single room without betting conditions.
3. Leaderboard: Displays top 9 winnings next to each player's username.

Pause Menu

The pause button is located in the top-right corner of the screen. Clicking it will freeze the game, display the player's current stats, and bring up a menu with the following buttons:

1. Resume
2. Restart

3. Quit: Brings the player to the main menu

Player

Controls: The classic W-A-S-D controls to move the player up-left-down-right. Aim and fire projectiles by pointing and left-clicking the mouse. Reload automatically by depleting all ammo or by pressing R.

Health: The red bar UI, located at the top middle of the screen, displays the player's health. It shrinks as the player loses health. When the bar is entirely black, the player loses the game. After clearing a room, the player regenerates health equivalent to the grunt's base damage stat. Every five rooms, the player's maximum health is incremented by 1.

Money: The $ symbol UI at the top-left of the screen displays the player's money. Players earn money through defeating enemies and completing levels under wager conditions. The final dollar amount represents the player's score.

Ammo: The Ammo UI text, located at the bottom left of the screen, displays the default amount of 10 bullets and the reloading action. The default reload time is 2.5 seconds.

Purchases and Gambling: Every time the player enters a new room, they are prompted with either the wager shop or the reward shop. The reward shop occurs every three rooms, and all other rooms are the wager shop. The player can purchase permanent or temporary wagers that make the game more difficult but grant them money at the end of the round. On the reward screen, the player can purchase permanent upgrades to their stats.

Enemies

All enemies can perform a standard damage amount. Grunt and Thief only damage the player by contact. Damage repeats for every second of sustained contact. Shooters can damage players through contact or by firing projectiles. Projectiles do standard damage. Shooters have a chance to perform critical hit damage upon contact. Shooters maintain their critical hit chance with sustained contact. Every five rounds, enemy damage and enemy health base stats increment. Each of the following three enemy types has different traits:

1. <u>Grunt</u>: Standard enemy type. Slowly moves toward the player. High health. No additional abilities.
2. <u>Thief</u>: Quickly moves around the room along a sinusoidal path. Low health. Its trait is stealing player money upon contact.
3. <u>Shooter</u>: Slowly moves toward the player. Medium health. Its trait is shooting projectiles that have a chance of critical hit damage upon contact (this chance slowly increases as the game progresses).

Win Conditions

There is no end state for the game. Arcade mode progressively gets more difficult until the player loses. However, win conditions exist to progress the player further into the game. Upon entering a room, the player is informed of that room's win condition (which is randomly selected when the player enters the room) before being prompted with the wager/reward screen. There are two possible win conditions:

1. <u>Time Room</u>: Upon entering the room, the player is prompted with a screen that says "Survive for: __". During the round, the Time text UI, located at the top-right of the screen, displays the remaining time until level completion. As the game progresses, the survival time for each Time Room will increase.
2. <u>Enemy Room</u>: Upon entering the room, the player is prompted with a screen that says "Defeat: __ enemies". The Enemies text UI, located at the top-right of the screen, displays the remaining enemies to defeat for level completion. As the game progresses, the number of enemies the player must defeat for each Enemy Room will increase.

Gambling (Wager Screen and Reward Screen)

1. Wager Types:

   Reverts after the room is completed:

   a. Enemy population *buff* (Decreases spawn interval by 0.5 seconds)
   b. Enemy health *buff* (Increases health of all enemies by 1)
   c. Player reload *debuff* (Increases reload time by 0.5 seconds)

   Permanent:

   d. Player ammo count *debuff* (Decreases ammo capacity by 1)
   e. Player health *debuff* (Decreases maximum health by 1)
   f. Player speed *debuff* (Decreases movement speed by 0.5)

   Note that there is a 20% chance that for any time the player sees the wager screen, one of the wager options will be a jackpot wager. A jackpot wager has 10x the rewards of a normal wager, incentivizing the player to go all in.

Reward System and Power-ups (Permanent):

a. Increased Player Health (Increases current health by 1)
b. Increased Max Health (Increases maximum health by 1)
c. Increased Ammo Count (Increases ammo capacity by 1)
d. Decreased Reload Time (Decreases reload time by 0.5 seconds)
e. Increased Speed (Increases movement speed by 0.5)

Technical Documentation

How to Add to the Code

Download GitHub Desktop (https://desktop.github.com/download/) and clone our code repository (https://github.com/JMBreard/CS370-Raging-Gambler) to create a local version. Then, download the Unity Hub (https://unity.com/download) and download the version of Unity we used to develop the game (Version 6000.0.37f1), which can be found on the Unity Download Archive (unity3d.com/get-unity/download/archive). From here, open the local version of the repository through Unity Hub by selecting only the "Raging Gambler" folder. Unity will be able to open the project with all of our code. You may now use the Unity Editor to make changes to the code, which will be reflected in GitHub Desktop. Create new branches through GitHub Desktop to develop new features or change code. Any new changes will be reflected in GitHub Desktop, which will allow you to push changes to the local repository before pushing to the server repository. For working on scenes, create a new test scene to change features, as GitHub Desktop will have issues merging branches and code together if the same scene is edited and other changes have been made before pushing.

General Design

The Raging Gambler is built around three main “managers” that act as the game’s control center: game manager, gamble manager, and reward manager. The game manager facilitates flow by sequencing each level’s progression. The gamble manager and reward manager facilitate game balance and enact the gambling mechanic through altering player and enemy stats. The gamble manager focuses on monetary rewards by betting on surviving a room’s conditions. The reward manager allows the player to use money to buy back health, increase speed, or boost the stats needed to compensate for room conditions. Although surviving longer helps, the path to the leaderboard is winning the most money.

This is a general overview of script interactions for player and enemy’s money and health changes. Player attributes are altered through the PlayerController and HealthController scripts. Enemy attributes are altered through the EnemyController, EnemySpawner, and HealthController scripts. HealthController communicates with the PlayerMoney script and implements the IDamageable interface directing health and damage control functions. Additionally, HealthController communicates with the Enemy script for Thief enemies to steal playerMoney and Shooter enemies to perform critical damage.

Architecture

Game Manager

<u>Non-Script Components:</u>

Four 2D Box Colliders: One collider just off-screen of each doorway to detect, using *OnTriggerEnter2D(Collider2D collision)*, when the player is in the next room.

<u>GameManager.cs</u>

1. <u>Room Generation</u>: Creates a new room connected to one side of the current room. A door opens into the new room. The camera pans to the new room when the player crosses its threshold. Handles enemy difficulty scaling and regenerates player health.
    a. *moveToNextRoom():* see *OnTriggerEnter2D(Collider2D collision)* for updated room variables.
        i. *KillAll()* destroys remaining enemies in the current room
        ii. *Increment()* raises time and enemy count as a function of level
            1. remainingTime += time_difficulty * level_counter
            2. enemiesNeeded += enemy_count_difficulty * level_counter

iii. currentDoorIndex represents the four doors within a room as an integer between 0 and 3. switch(currentDoorIndex) randomly chooses a door to open as long as it is not the door the player came from. nextRoom will align with this door based on newPos.x or newPos.y.

iv. *NewRoomObstacles(newPos)* populates the new room with obstacles. It is a variation of *SpawnObstacles()* that generates obstacles in new rooms with a random amount and placement.

v. WagerCounts marks the selected wager by multiplying it by 1 and all others by 0. When the wagers are looped through, only the marked reward value is added to playerMoney. Additionally, WagerCounts will debuff enemies according to wagers purchased.

vi. *ScaleEnemies()* is called every 5 levels to increase enemy health and damage output by 1.

   1. For balance, player max health increments by 1 every 5 levels.

vii. *HealthRegen()* rejuvenates player health at the end of each level.

   1. Health rejuvenated = enemy's base damage stat (dmg_ctr)
   2. Does not exceed player max health

2. Win Conditions: Either enemy or time-based. The player must survive for a set amount of time or defeat a set number of enemies.
   a. *pickRoomCondition()* is triggered when the player enters a room. It randomly chooses the win condition and formats its associated UI text.
   b. *Update()* decrements timeRoom's remainingTime. When time runs out, the associated UI text is deactivated, as well as the enemySpawner. *moveToNextRoom()* is triggered.
   c. *EnemiesLeftUpdate()* is activated when the player enters a room and is triggered from the *HealthController* script when an enemy dies. enemiesNeeded decrements and the associated UI text is updated.
      i. *Win()* activates when enemiesNeeded = 0. All enemies and the room type are set to false then *moveToNextRoom()* is called.
3. Menu Handling: Triggers room condition prompt, the gamble manager/wager shop, game over screen, and title screen with leaderboard.
   a. *GameOver()* destroys all enemies, activates the UI, and displays the player's final money amount as the score.
   b. *Restart()* loads the scene manager's Title Scene or Leaderboard Title Screen. The scoreManager prompts the player with playerMoney and a userName entry.
   c. *moveToShop()* is triggered through the *ShopTrigger* script assigned to the door objects. It activates the WagerGenerator UI.

    d. *isMouseOverUIIgnore()* senses mousePosition as an input to determine if it is hovering over a UI using raycasting. If it is not, then it is ignored. This prevents the mouse from triggering other events outside of the opened menu.

4. Housekeeping: Increments win conditions, rewards player's wager winnings, destroys current room enemies, resets obstacles, and restarts game to leaderboard with sound handling.

<u>CrosshairController.cs</u>

The crosshair controller script makes the Dynamic Crosshair Game Object replace the default OS mouse whenever the player is not hovering over UI (such as the pause menu) using raycast detection. The script also makes the crosshair flash red every time the player shoots.

- Awake() hides the OS cursor (Cursor.visible = false, Cursor.lockState = None), assigns defaultSprite and defaultColor to the serialized SpriteRenderer.
- Every frame Update() checks: EventSystem.current.IsPointerOverGameObject().
    - True: Shows the system cursor and disables crosshairRenderer.
    - False: Hides the system cursor, enables the sprite, then positions it at Camera.main.ScreenToWorldPoint(Input.mousePosition).
- If the pausePanel is active (pausePanel.activeSelf), the crosshair sprite stays hidden so it never sits on top of the menu UI.
- Shoot flash feedback – OnEnable() subscribes to PlayerController.OnShoot; handler starts the FlashCrosshair() coroutine:
    - Swap to shootSprite and shootColor.
    - yield return new WaitForSeconds(flashDuration) (0.01 - 0.5 s via inspector).
    - Revert to the default sprite and color. Any running coroutine is first cancelled by StopAllCoroutines() to avoid overlap.

Gamble Manager

1. Wagers: Creates a kind of shop that offers temporary and permanent debuffs under which to play. If the player survives the round, then a reward is collected, and temporary debuffs expire.

   a. *Start()* instantiates the wager shop as an array of items that contain the following elements: name, cost, reward, and image. The image has an event listener that enables a wager upon clicking.
   b. *BuyWager()* is called when a wager is clicked. It subtracts the cost from playerMoney and applies the condition.
   c. *ApplyWager()* is called when a wager is clicked through *BuyWager()*. switch(wager.name) checks which wager was selected and triggers the function, communicating with the appropriate script. This will change the desired player or enemy stat (interactions described in the general design).
   d. *UpdateWagers*() updates costs and rewards as a function of level. There is a 20% chance that a wager shop will generate rewards at 10x scale for a type of jackpot functionality.
      i. Scale = 1.4 (20% chance of being 10)
      ii. Cost = base_cost * (1 + (levelCounter * 1.1 / 10))
      iii. Reward = base_reward * (1 + (levelCounter * scale / 10))
   e. *ScaleEnemies()* increments enemy health and damage by 1. Dmg_ctr increments by 1.
      i. Dmg_ctr is used to scale enemies and calculate health regeneration.

Reward Manager

<u>RewardManager.cs</u>

1. Rewards: Creates a shop that offers the player stat buffs at a high price. Rewards are significantly more expensive than wagers, so players cannot overbuy and make the game easy.
   a. *Start()* instantiates the reward shop as an array of items that contain the following elements: name, cost, and image. The image has an event listener that enables a wager upon clicking.
   b. *BuyReward()* is called when a reward is clicked. It subtracts the cost from playerMoney and applies the condition.
   c. *ApplyReward()* is called when a reward is clicked through *BuyReward()*. switch(reward.name) checks which reward was selected and triggers the function communicating with the appropriate script. This will change the desired player stat (interactions described in general design).
   d. *UpdateRewardCosts*() updates reward costs as a function of level.
      i. Cost = base_cost * (1 + (levelCounter * 1.5 / 10))
   e. *HealthRegen()* regenerates the player's currentHealth by the enemy's base damage count, which is tracked using dmg_ctr.

Player

Non-Script components:

1. RigidBody2D (allows Unity physics engine to act upon the player)
2. Capsule Collider 2D (enables collision detection)
3. Animator (handles animations)

1. Controls: Player direction is defined by cursor position in the PlayerRotation.cs script. Player movement is defined by W-A-S-D controls, the reload key is R, and the player's movement speed, all of which are controlled by the PlayerController.cs script.
    a. *reduceSpeed()* is a GambleManager function that reduces the player's movement speed.
2. Projectiles and Ammo: PoolManager.cs script instantiates the bullet prefab and allows the player to fire. PlayerController.cs script handles firing, reloading, and ammunition.
    a. PoolManager.cs
        i. *GenerateBullets()* instantiates the desired number of _bulletPrefab game objects inside the pool manager. The pool manager reduces clutter within the sample scene. No bullets are active.
        ii. *RequestBullet()* loops through the _bulletPool for inactive bullets to return to the player. If none exist, then a newBullet prefab is instantiated in the pool for the player to fire.
    b. PlayerController.cs
        i. *Fire()* decrements _currentAmmoCount and then calls RequestBullet(). The bullet prefab position is set in front of the player, and its trajectory vector is set according to player rotation (direction as defined in PlayerRotation.cs script).
        ii. *Reload()* is an IEnumerator coroutine that makes false _canFire and reloading booleans and forces the player to WaitForSeconds(reloadTime). The default is 3 seconds. Booleans are reset to true.
        iii. *OnEnable()* instantiates the player and initializes starting values. 10 bullets for _currentAmmoCount.
        iv. *OnDisable()* importantly disables references to destroyed objects, precluding null reference errors.
        v. *increaseReloadTime()* and *decreaseMaxAmmoCount()* are GambleManager functions that increase player reload time and decrease player max ammo count.

3. Health: Since the player deals damage using bullets, the ProjectileMovement.cs script defines the IDamagable interface with the following functions: *Health { get; set; }*, *Damage()*, and *TakeDamage(int amount)*. This is implemented by the HealthController.cs script which triggers HealthBar changes.
    a. ProjectileMovement.cs
        i. *OnTriggerEnter2D()* damages the game object that the bullet comes into contact with. The bullet is hidden.
        ii. *Hide()* deactivates the bullet after 1 second or a collision.
    b. HealthController.cs
        i. *Damage()* is defaulted to 1 health by calling *TakeDamage(1)*.
        ii. *TakeDamage()* is called by *Damage()* and other mechanisms like critical hit. If a player is not dead, decrement currentHealth. If currentHealth is 0, then trigger *Die()*. Update the health bar.
        iii. *Die()* will give the player $5 for killing an enemy and will deactivate obstacles or destroy game objects whose health is 0.
        iv. *reduceMaxHealth()* and *increaseMaxHealth()* are GambleManager functions that reduce player max health and increase enemy max health.
4. Money: PlayerMoney is the score. Money is used to purchase wager conditions and player stat boosts in the Reward Shop. PlayerMoney cannot be less than 0.
    a. *Start()* initializes playerMoney to $100
    b. *Steal()* is an Enemy function that steals money from the player when a thief makes contact.
        i. playerMoney -= level_counter * 2
    c. *addMoney()* is called to adjust the player's money. This is done by the Wager Shop and upon enemy death.
    d. *subtractMoney()* is called to adjust the player's money. This is done by the Wager Shop, Reward Shop, and thief enemy types.
    e. *UpdateMoneyText()* updates the UI display at the top-left of the screen.

Enemy Spawner

EnemySpawner.cs script creates enemies centered around the player, but not on the player. It provides some functionality like adjusting spawn rates and enemy health (level scaling and wagers).

a. *SpawnEnemy()* will randomly spawn enemies from its enemySpawnData array, which is made up of instances of an EnemySpawnData struct. Each object of the struct includes an enemyPreab and its corresponding spawnChance. The selected enemySpawnData's enemyPrefab is spawned a set distance from the player (using a 2D unit vector given a

random direction and a magnitude of spawnDistance, which is set to 10 to keep enemy spawning within the room's bounds). The method then performs a comparison so that its spawn position will recalculate if it is less than a minimum distanceFromPlayer (set to 4) so that no enemies spawn too close to the player.
    i. Grunt spawn chance = 50%
    ii. Thief spawn chance = 30%
    iii. Shooter spawn chance = 20%

b. *Update()* calls *SpawnEnemy()* at an interval defined by spawnInterval

The following methods are all called by <u>GambleManager.cs</u> when the player makes wagers or prucahses rewards that impact enemy health:

c. *StopSpawning()* disables the isSpawning boolean to deactivate enemy spawning upon level completion.
d. *increaseSpawnRate()* and *decreaseSpawnRate()* adjust the default spawn rate of 2 seconds by 0.5 second increments.
e. *addEnemyHealth()* and *subtractEnemyHealth()* adjust the enemy health base stat by + or - 1.
    i. Grunt base health = 3
    ii. Shooter base health = 2
    iii. Thief base health = 1
f. *GetSpawnRate()* and *GetEnemyHealthBuff()* expose the current interval and health bonus.

Enemies

The three enemy prefabs each share several components with each other and the player:

4. RigidBody2D (allows Unity physics engine to act upon enemies)
5. Capsule Collider 2D (enables collision detection)
6. HealthController
7. Animator (handles animations)

Grunt (Default Enemy)

Controller: <u>EnemyController.cs</u> provides damage and movement functionality, as well as defining special traits.

1. *FixedUpdate()* defines enemy movement toward the player's position. If an enemy encounters an obstacle, then it will move perpendicular to its current vector until it clears the obstacle. Also calls *FlipSprite().*
2. *OnCollisionEnter2D()* instantiates playerHealth and playerMoney so that enemies can deal damage and steal as defined by its traits.
3. *OnCollisionExit2D()* ends the enemy's damage coroutine.

4. *SustainedDamage()* compares game time to dmgTimer to deal damage for every 1 second of sustained enemy contact
5. *Trait()* calls the *Steal()* and *CriticalHit()* functions for thief and shooter enemy types (stealing is defined in the PlayerMoney.cs script)
6. *CriticalHit()* deals damage to player health in proportion to the current level (currently 1 to 1, but an adjustable multiplier exists). The chance of a critical hit activating scales with level but caps at 33% on level 17.
7. *FlipSprite()* takes the enemy's current movement direction as a parameter and flips the enemy's sprite along the x-axis to match the direction it's moving in.

Thief (CurvedEnemy)

The Thief enemy uses the CurvedEnemy.cs script, which inherits from the EnemyController.cs script instead of using EnemyController.cs. The only overridden method is *FixedUpdate(),* which is modified so that when the enemy moves toward the player, it adds a perpendicular offset equal to sin(Time.time * curveFrequency) * curveAmplitude, producing an oscillating sideways deviation. The resulting vector is normalised and fed to rb.MovePosition, so the enemy drifts toward the player on an oscillating path whose width and speed are defined by curveAmplitude and curveFrequency.

Shooter (ShooterEnemy)

The Shooter enemy uses the ShooterEnemy.cs script, which likewise inherits from EnemyController.cs. Its only overridden method is *FixedUpdate()*; after calling the default EnemyController movement, it increments shootTimer and, whenever shootTimer ≥ shootRate, resets the timer and calls *ShootAtPlayer()*. *ShootAtPlayer()* instantiates the projectilePrefab at the enemy's current position, calculates the normalised vector toward the player, rotates the projectile to face that direction, and assigns direction * projectileSpeed to its Rigidbody2D.linearVelocity.
This enemy chases the player just like the default enemy, but also periodically fires projectiles at a rate and speed defined by shootRate and projectileSpeed.

Boss Feature Documentation (in repository but not in main branch)

Gameplay:

Boss Rooms
Boss Encounter: Every 10 levels, the player faces a unique boss battle instead of regular time or enemy-based rooms. The "Boss" text appears at the top-right of the screen, indicating a boss room.
Win Condition: To complete a boss room, the player must defeat the boss enemy. Boss minions do not count toward the win condition.
Rewards: Boss battles provide significant monetary rewards - defeating a boss yields 200 money, with additional rewards when bosses transition between phases.

Boss Types
Basic Boss (SimpleBoss): A larger, stronger enemy with 500 health that slowly pursues the player. It periodically spawns minions to assist in the battle.
Advanced Boss (BossEnemy): A multi-phase boss with 1000 health that changes attack patterns and appearance as its health decreases. Features more complex attack patterns, including projectiles and AOE attacks.

Boss Mechanics
Boss Health: Bosses have significantly more health than regular enemies (500-1000 HP).
Health Bar UI: A special boss health bar appears at the top of the screen during boss encounters, showing the boss's name and remaining health.
Phases: Advanced bosses transition through multiple phases (at 60% and 30% health), changing color, size, and attack patterns with each phase.
Minion Spawning: Bosses periodically spawn minion enemies to assist them in battle.
- Minions have 2 health points and die in two hits
- Minions are marked as "boss minions" and don't affect the room's win condition

Boss Attacks
Basic Movement: All bosses move toward the player, similar to standard enemies but typically slower.
Projectile Attacks: Advanced bosses fire projectiles at the player.
Spread Attacks: Advanced bosses can fire multiple projectiles in a spread pattern.
AOE Attacks: Advanced bosses can perform area-of-effect attacks, firing projectiles in all directions.
Minion Summoning: Bosses periodically spawn minion enemies to assist them in battle.

Technical Implementation:

Boss System Architecture

Boss Spawning: The GameManager handles spawning the boss after a delay when a boss room is created (every 10 levels).
Boss Health: Both boss types implement the IDamagable interface to handle damage and health mechanics.
Boss UI: A dedicated BossUI script manages the boss health bar and name display.
Phase Management: The BossEnemy script handles phase transitions, changing appearance and attack patterns based on health thresholds.

Boss Room Management
Room Identification: The GameManager sets a "bossRoom" flag when creating a boss level.
Win Condition: Boss rooms ignore the standard enemy counter and time mechanics, focusing solely on boss defeat.

Boss Defeat: When a boss is defeated, it calls GameManager.BossDefeated() which triggers the completion of the room.

Minion Management
Minion Spawning: Bosses periodically spawn minions to assist in the battle.
Minion Properties: Minions are flagged with isBossMinion = true to prevent them from affecting the room's win condition.
Minion Death: When minions die, the EnemiesLeftUpdate() method is not called due to the isBossMinion flag, ensuring only the boss's defeat triggers the win condition.

Library Dependencies:

- UnityEngine – core engine API (GameObject lifecycle, transforms, physics, audio, etc.).
- UnityEngine.UI – built-in UI system: Canvas, Image, Button, Slider, and related components.
- TMPro – TextMesh Pro package for crisp, richly styled in-game text.
- System.Collections – non-generic collections (ArrayList, Queue) and the IEnumerator interface.
- System.Collections.Generic – generic collections such as List<>, Dictionary<>, HashSet<>.
- UnityEngine.EventSystems – event / ray-cast layer that drives UI clicks, pointer detection, and EventSystem utilities.
- UnityEngine.SceneManagement – scene loading, unloading, and additive scene operations.

- Unity.Collections – high-performance, native containers (NativeArray, NativeList) usable with Burst jobs.
- System – fundamental C# types and language constructs (Math, Random, exceptions, attributes, etc.).

References:

1. Connecting GitHub to Unity:
    a. https://www.youtube.com/watch?v=qpXxcvS-g3g
2. Creating Basic 4D Movement:
    a. https://www.youtube.com/watch?v=fcKGqxUuENk&t=19s
3. Creating Projectiles:
    a. https://medium.com/nerd-for-tech/2d-player-shooting-mechanics-in-unity-6fda9c8e92fd#d0bb
4. Creating Obstacles:
    a. https://www.youtube.com/watch?v=dpxPc3t3kR8
5. Unity Basic Tutorial:
    a. https://www.youtube.com/watch?v=XtQMytORBmM
6. Create Pause Menu:
    a. https://www.youtube.com/watch?v=MNUYe0PWNNs
7. Changing Scenes:
    a. https://www.youtube.com/watch?v=3SdMFPdSi7M
8. Health and Damage:
    a. https://www.sharpcoderblog.com/blog/adding-health-system-in-unity-game
9. Health Bar:
    a. https://www.youtube.com/watch?v=0T5ei9jN63M
10. Money System:
    a. https://www.youtube.com/watch?v=XJlPF4GtydU
11. Gambling System UI:
    a. https://www.youtube.com/watch?v=aWd17lOdxJs
12. Sliding Camera:
    a. https://www.youtube.com/watch?app=desktop&v=PA5DgZfRsAM&t=0s
13. Countdown Timer:
    a. https://www.youtube.com/watch?v=POq1i8FyRyQ
14. Mouse Over UI:
    a. https://www.youtube.com/watch?v=ptmum1FXiLE
15. Leaderboard UI:

   a. https://www.youtube.com/watch?v=-O7zeq7xMLw
   b. https://www.youtube.com/watch?v=iAbaqGYdnyI

Future

The team achieved its initial goal of implementing a working 2D rogue-like dungeon crawler with gambling-based mechanics. We built the framework to be easily expandable due to its modular system (as described in general architecture). Overall, we are happy with the result. However, there are still so many things that can be done to make it even better. Here are some of our ideas.

1. A survival mode could easily be created with a couple of tweaks to the tutorial and arcade mode. The scoring system should be time-based rather than money-based. The player should only exist in one room and the reward shop should appear every one to two minutes. Enemy scaling should be time-based so that its health and damage increment alongside the reward shop's appearance. We found that people really enjoyed the tutorial-style gameplay, and this could be a fun way to expand it.
2. Boss System mechanics were implemented in a branch outside of main branch. We did not have an opportunity to integrate the system into the game. This could be another fun way to reap huge winnings. We envisioned a boss fight every ten levels. There are two approaches for winning or losing these. The first approach is that losing ends the run, but winning gives a huge payout. The second is that dying would not end the game, but simply continue to the next level. In the second case, the boss fight could be an opportunity to reap interesting rewards like stat boosts or different weapons that increase survival chances.
3. In terms of aesthetics, we imagined a casino floor with roulette tables, slot machines, blackjack, etc. We had a couple of designs, but found that it impeded player movement. Due to time and how well the initial small room design worked, we reverted to a simple layout. However, it should be possible to have a layout with all of the typical casino elements while providing the player sufficient space.
4. Feature List TODO (specific):
   a. Projectile types:
      i. Bird shot: spread of 3-5 bullets like a shotgun
      ii. Burst fire: volley of 3-5 bullets in quick succession
      iii. Auto: press and hold the left mouse button to rapid fire.
      iv. Grenade: Deals area damage, affecting multiple enemies
      v. Aesthetic: Options for projectiles look like poker chips, playing cards, etc.

b. Obstacle types: Shooting different obstacle types should give money based on chance. This would be another decision point because obstacles are useful for avoiding enemies, but could also raise the player’s score. We only use slot machines, but it could be fun to implement other elements like poker tables, roulette, and craps (dice throwing).
c. Dodge: The player should have a roll or dash mechanic to quickly evade enemies. This should have a recharge rate (3-5 seconds) and could either parry all damage (0.5 second invincibility) or take one enemy’s base stat damage. It should not allow the player to be damaged per enemy contact, otherwise, the evasion functionality is diminished.
d. Camouflage: The carpet colors should randomly change for each room. It should toggle between red, yellow, and blue to match the enemy colors. Currently, only the blue carpet is implemented, which camouflages the shooter enemy type.