Custom Project Document

Swin-Towers

Show Wai Yan

Student ID: 105293041

Course: Computer Science (Software Development)
Unit Code: COS10009
Unit Name: Introduction to Programming

Swinburne University of Technology

August 2024

Project link: Click Here
Video link: Click Here
GitHub Repo Link: Click Here

Abstract

This document presents the design, development, and evaluation of **Swin Towers**, a 2D tower defense game created using Ruby and the Gosu library. The game challenges players to defend a virtual town from waves of monsters, using strategically placed towers. Players must manage resources efficiently to build and upgrade towers, ensuring the town's survival. The project focuses on integrating game mechanics, such as resource management and dynamic enemy behavior, while maintaining a visually appealing and interactive user experience. This research

highlights the development process, technical challenges, and innovations that contribute to an engaging gaming experience.

Introduction

Tower defense games have been a staple in the gaming industry, offering players a mix of strategic planning and resource management. **Swin Towers**, a 2D tower defense game developed using Ruby and the Gosu library, is an innovative addition to this genre. The game's primary objective is to protect a fictional town from waves of incoming monsters by building and upgrading towers strategically.

The development of **Swin Towers** explores the intersection of game design principles and programming techniques. The game incorporates diverse gameplay elements, such as monsters, resource allocation challenges, and progressive difficulty levels. By leveraging the Gosu library, the project demonstrates how a lightweight framework can be used to create engaging and visually interactive applications.

This document provides a comprehensive overview of the game's design and development. It covers the implementation of core mechanics, the challenges faced during development, and the solutions applied to overcome them. Additionally, the document evaluates the game's performance and its potential for further enhancements. **Swin Towers** not only serves as an entertainment product but also as a learning opportunity to explore game development techniques and resource optimization.

Design and Development

The design and development of **Swin Towers** focused on creating an engaging tower defense game with strategic resource management and

intuitive gameplay. This section outlines the key design choices, technical architecture, and development process that shaped the game.

1. Game Design Overview

Swin Towers is a classic tower defense game where players build and upgrade towers to protect a town from waves of invading monsters. Key design elements include:

- **Towers and Upgrades**: Players strategically place towers with unique abilities and can spend resources (diamonds) to enhance their performance.
- **Enemies with Unique Traits**: Monsters vary in speed, strength, and durability, requiring players to adapt their strategies.
- **Resource Management**: Players balance spending resources between building new towers and upgrading existing ones.
- **Win/Loss Conditions**: The game ends when either all waves are cleared (win) or a specific number of enemies enter the town (loss).

2. Technical Architecture

The game was developed using the **Gosu** library, a Ruby-based framework for 2D games. The architecture consists of:

- **Modules and Classes**:
  - **Game Class**: Manages the main game loop, including updates, rendering, and state transitions.
  - **Tower Class**: Represents individual towers, including their position, attack logic, and upgrade mechanics.
  - **Enemy Class**: Handles enemy movement, interactions, and behaviors.
  - **UI Components**: Separate classes for buttons, fonts, and other user interface elements.
- **Z-Order Layers**: Defines rendering priorities for background, towers, enemies, and UI elements.

- **Assets Management**: Utilizes external files for images, sounds, and wave configurations, ensuring modular and scalable resource handling.

3. Development Process

The game was developed in iterative stages:

- **Prototype Phase**: Established the core gameplay mechanics, including tower placement, enemy movement, and wave progression.
- **Design Phase**: Created game assets such as background maps, sprites for towers and enemies, and UI elements.
- **Implementation Phase**: Integrated assets with the Gosu framework and implemented features like resource management, upgrades, and win/loss conditions.
- **Testing and Debugging**: Iteratively tested the game to refine gameplay balance, fix bugs, and improve performance.

4. Challenges and Solutions

Several challenges arose during development:

- **Frame Synchronization Issues**: Early versions experienced sprite frame jumping in animations. This was resolved by implementing a consistent frame update mechanism using modulus arithmetic.
- **Resource Balancing**: Ensuring that diamond allocation was neither too lenient nor too strict required repeated playtesting and adjustments.
- **User Interface**: Creating an intuitive UI required careful placement of buttons and clear feedback mechanisms to avoid player confusion.
- **Enemy Movement Issues**: Enemies failed to follow their paths correctly, resulting in erratic movement or straying from the map. A waypoint system was used to guide enemies along predefined paths.

Each enemy calculated its movement vector to the next waypoint, ensuring smooth and accurate navigation.

- **Rotation of Objects**: Towers needed to rotate to face enemies, but the initial implementation caused incorrect angles or jerky movement. The rotation angle was calculated using trigonometric functions based on the relative positions of the tower and the enemy. Smooth interpolation ensured the rotation appeared fluid and natural.
- **Highlight Tower Feature and Automatic Movement**: Towers needed to be highlighted when selected, and during placement, they had to automatically adjust to valid positions on the map. The highlight feature was implemented by adding a visual overlay. For placement, towers checked nearby valid grid spots and automatically snapped into the nearest valid position if placed incorrectly.
- **Collision Detection**: Detecting collisions between projectiles, enemies, and towers required precise calculations and caused performance issues with many objects.
- **Enemy Detection**: Towers needed to detect enemies within their range and prioritize targets effectively.
- **UI State Changing**: Transitioning between different UI states (e.g., start menu, in-game, and game over) caused inconsistent behaviors and broken buttons. A state management system was implemented, where each state (e.g., start menu, in-game) was encapsulated in a distinct class or method group. Buttons and UI elements were activated or deactivated based on the current state, ensuring smooth transitions.

5. Gameplay Balancing

Playtesting was integral to balancing enemy waves, tower costs, and upgrade effects. Player feedback helped refine difficulty progression, ensuring a fair challenge while maintaining engagement.

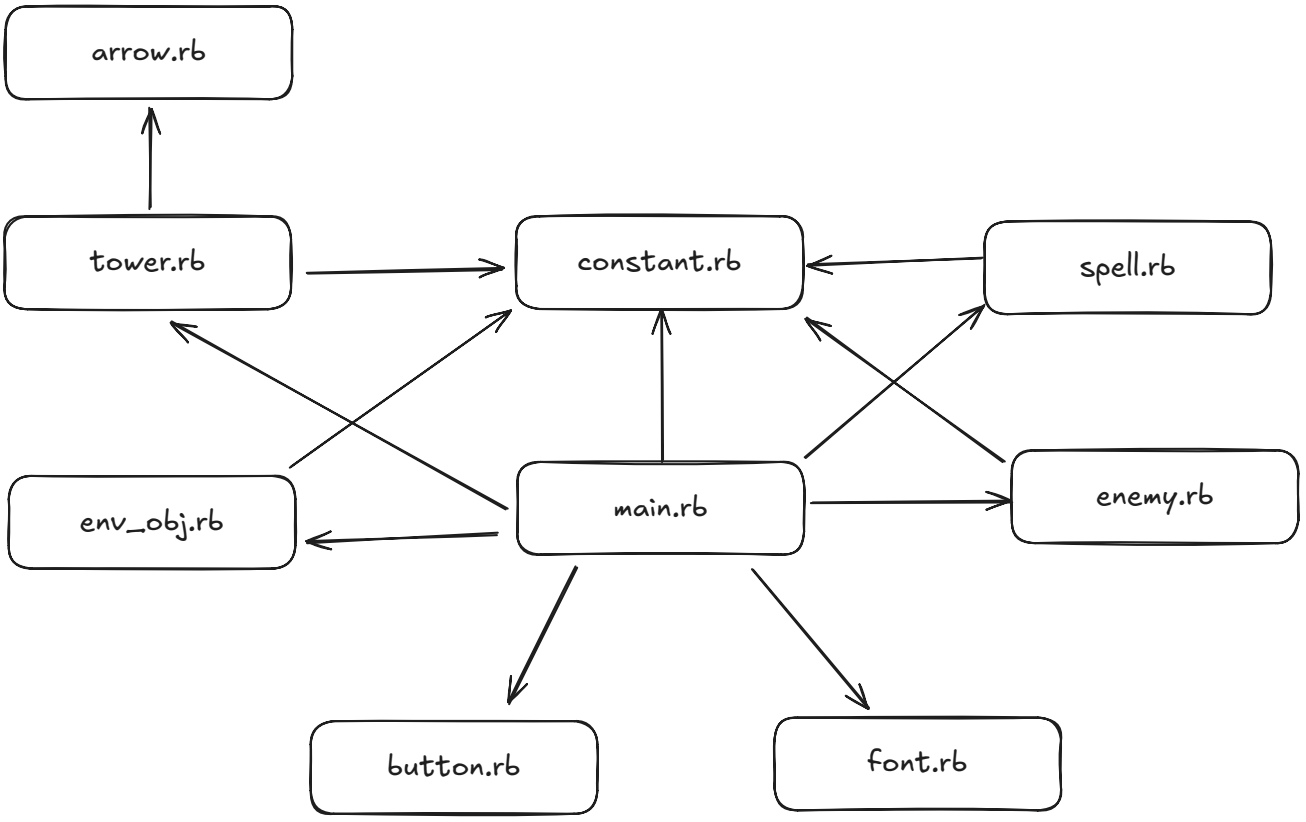
By leveraging principles of game design and programming, **Swin Towers** successfully combines strategic gameplay, smooth mechanics, and a visually appealing interface, offering an immersive experience for players.

Implementation Details

Overview of the Main Program Procedure

The provided code represents the core logic of a tower defense game, implemented using the **Gosu** framework in Ruby. Below is an overview of the program structure and how it handles various aspects of the game.

Main Structure and Flow



1. **Game Class:**
    - The *Game* class inherits from *Gosu::Window*, making it the main entry point for the game. It defines the window's size, title, and controls the game's flow, including initialization, updates, and rendering.
2. **Initialization (*initialize* and *setup_game*):**
    - The *initialize* method sets up the screen dimensions and window title. The *setup_game* method initializes the game state,

including enemies, towers, and the user interface (UI).
    - Resources such as background images, music, and sound effects are also loaded during setup.
3. **Game Flow Control:**
    - The game operates in different **UI states**: start menu, in-game, and end menu. Each state has its own set of buttons and actions.
    - The game loops through various states, managing transitions between them based on user actions (e.g., start button click, wave completion).
4. **Enemy and Tower Management:**
    - **Enemies**: The game spawns enemies based on wave data. Each enemy moves along a predetermined path, and their interaction with the game world is handled by *Enemy* objects.
    - **Towers**: The game allows players to place towers at specific locations. Towers can be upgraded and targeted toward enemies automatically. Interaction with the tower objects is handled via mouse input.
5. **User Interface (UI):**
    - The UI dynamically updates based on the game's state, displaying information like heart count, diamond count, and the current wave.
    - The UI also includes buttons for actions like creating/upgrading towers, casting spells, starting waves, and pausing/ending the game.
6. **Game Update Cycle (*update* method):**
    - Each frame, the game updates the status of all game elements:
        - **UI elements** are updated (buttons, texts).
        - **Enemies** are spawned and updated, including checking if they reach the end of their path.
        - **Towers** are updated, including targeting and attacking enemies.
        - **Spells** are cast and affect enemies.

        - The game checks if the game over or game finished conditions are met.
7. **Drawing the Game (*draw* method):**
    - Based on the current state (start menu, in-game, or end menu), the corresponding elements (background, towers, enemies, UI) are drawn to the screen.
8. **Input Handling (*button_down* method):**
    - This method responds to mouse and keyboard input:
        - Left-click is used for selecting and placing towers, casting spells, and interacting with UI elements.
        - Right-click is used to cancel tower placement or cancel spell overlays.

Key Features:

1. **Enemy Movement:**
    - Enemies spawn and follow predefined paths. The movement and interactions with the environment are handled via the *Enemy* class.
2. **Tower Mechanics:**
    - Towers can be created, upgraded, and targeted to attack enemies within range. Towers interact with the environment and enemies via mouse clicks and UI interactions.
3. **Spell System:**
    - Players can cast spells, such as a lightning spell, to damage enemies in range. The spell is managed via the *Spell* class, with its own overlay and effects.
4. **Resource Management:**
    - Players earn diamonds for defeating enemies and use them to create or upgrade towers. Hearts are used to track the player's health, which decreases if enemies reach the path's end.
5. **Wave System:**

- Enemies are spawned in waves. The game progresses through waves of increasing difficulty, with a special boss wave at the end.

6. **UI Management:**
    - The UI dynamically updates based on game state, displaying necessary information (e.g., resources, tower upgrades, current wave) and interacting with the player through buttons.

Important Concepts

Animation

In **Gosu**, animations are typically created by using a **sprite sheet**, which is a single image containing multiple frames. The frames are separated into tiles, which are then displayed one by one to create the animation effect.

- **Sprite Sheets**: A sprite sheet is a single image with multiple frames arranged in a grid. Each frame corresponds to a specific action or state in the animation (like running, jumping, etc.).
- `Gosu::Image.load_tiles`: This method loads the sprite sheet into multiple smaller images (tiles), each representing one frame.
- **Frame Computation**: To animate, the game cycles through these frames. The frame index changes over time, often with a fixed delay between each frame.

Example Code

```
class AnimatedSprite
  def initialize(window, sprite_sheet, width, height,
frame_delay)
    @frames = Gosu::Image.load_tiles(window, sprite_sheet,
width, height, retro: true)
    @frame_count = @frames.size
```

```
    @current_frame = 0
    @last_update = Gosu.milliseconds
    @frame_delay = frame_delay
  end

  def update
    if Gosu.milliseconds - @last_update > @frame_delay
      @current_frame = (@current_frame + 1) % @frame_count
      @last_update = Gosu.milliseconds
    end
  end

  def draw(x, y)
    @frames[@current_frame].draw(x, y, 1)
  end
end
```

Concept Explained:

1. **Sprite Sheet Loading**:
   - A sprite sheet is loaded into individual tiles using `Gosu::Image.load_tiles`, each tile representing a frame of animation. The `retro: true` flag ensures pixel-perfect rendering, ideal for pixel-art style games.
2. **Frame Index**:
   - The `@current_frame` keeps track of which tile (or frame) to draw.
   - Every time the frame is updated, it increments the frame index. Once the last frame is reached, it loops back to the first frame (`@current_frame = (@current_frame + 1) % @frame_count`).
   - We can also use `@current_frame = ( Gosu: :millisecond / @frame_delay) % @frame_count`
3. **Frame Delay**:

- The speed of the animation is controlled by `@frame_delay`. This is the time between frame updates, typically measured in milliseconds. The higher the delay, the slower the animation.

4. **Update and Draw**:
    - `update`: The `update` method checks if enough time has passed (based on `@frame_delay`) to switch to the next frame.
    - `draw`: The `draw` method renders the current frame at the specified position.

Animation Process:

- **Frame Cycling**: Each frame of the animation is drawn one at a time in a loop, creating the illusion of movement.
- **Timing**: The delay between frame updates controls the animation speed. A shorter delay results in faster animation, while a longer delay makes the animation appear slower.

This method of animation is widely used in 2D games for smooth, frame-by-frame sprite-based animations (e.g., walking, running, shooting).

Enemy Movement Using Vectors

In the game, enemy movement is implemented using **vectors** to ensure precise positioning and smooth movement. The concept of vectors is critical because it allows for efficient and flexible handling of movement and direction, offering simple operations like addition, subtraction, and normalization. These operations are intuitive and directly related to how game objects move in a 2D environment.

Why Use Vectors?

A vector is a mathematical entity that defines both **direction** and **magnitude** (or length). For a game character or enemy, a vector can

represent a position in the 2D world, and also the direction of movement. Vectors offer several advantages in game development:

- **Precision**: Vectors allow precise control over an enemy's movement, both in terms of position and direction.
- **Vector Operations**: You can perform arithmetic operations (addition, subtraction, etc.) directly on vectors, making it easy to manipulate positions and calculate movement.
- **Consistency**: The same syntax applies to both `x` and `y` coordinates, allowing simple operations like $Vector1 - Vector2$, which is equivalent to moving in a straight line from one point to another.

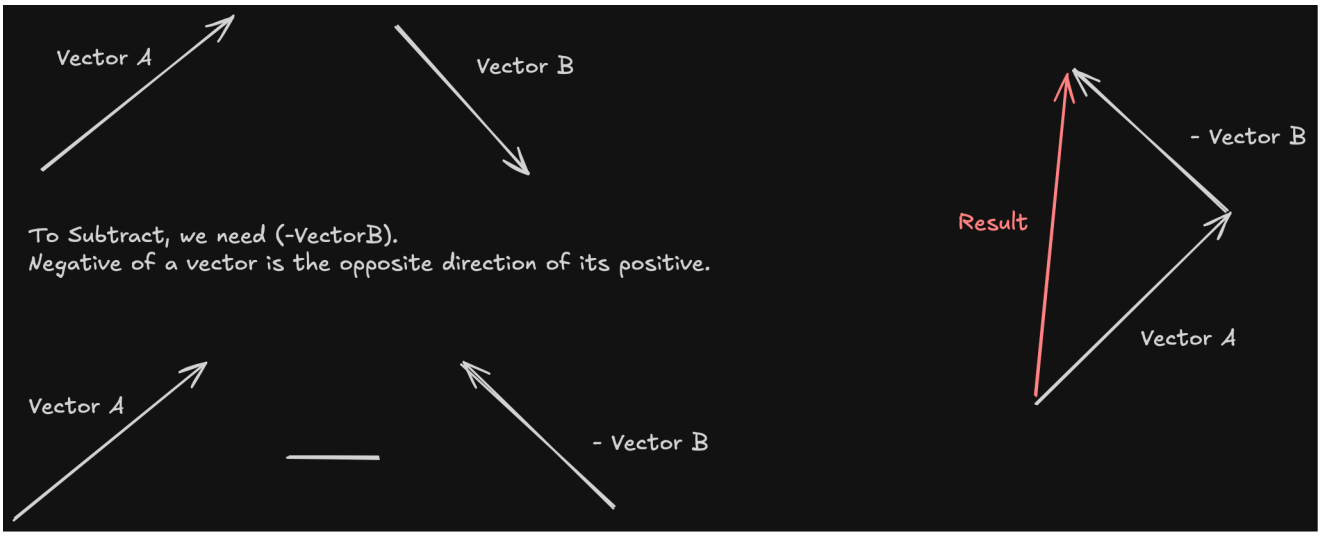
How Does Enemy Movement Work?

The basic idea behind enemy movement is to guide the enemy from its **current position** to the **target position**. The movement involves the following steps:

1. **Calculate the Direction**: Subtract the enemy's current position vector from the target position vector to get the direction.
2. **Normalize the Direction**: Normalize the resulting direction vector to make sure the movement speed remains consistent, regardless of the distance.
3. **Update the Position**: Add the normalized direction vector to the current position, moving the enemy closer to the target.

1. Subtracting Vectors to Find the Direction

The first step is to find the **direction** in which the enemy should move. This is done by subtracting the current position vector from the target position vector.



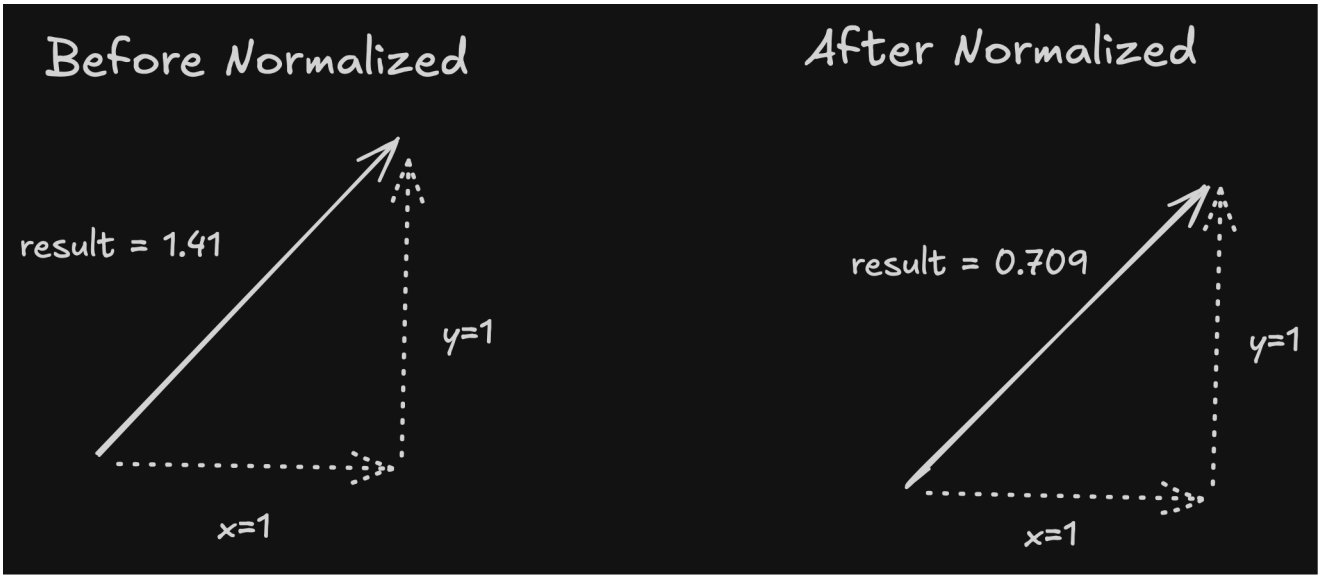
- **Formula**:

$$Direction = TargetPos - CurrentPos$$

The result of the subtraction gives a vector that points in the direction the enemy needs to move. The direction vector will have both **x** and **y** components that tell the enemy how to adjust its position.

2. Normalizing the Direction Vector

Once we have the direction, we need to **normalize** the vector. A **normalized vector** is a unit vector, meaning it has a length of **1**. This ensures that no matter how far the target is, the enemy moves at a consistent speed.



- **Why Normalize?**
  Without normalization, the movement would vary based on the distance between the enemy and the target. For example, if the enemy is far from the target, the direction vector's length would be larger than 1, resulting in faster movement. Normalizing the vector guarantees consistent movement speed.
- **Formula for Normalization**:

$$NormalizedVector = \frac{Direction}{LengthofDirection}$$

or

$$\begin{bmatrix} \frac{x}{result} \\ \frac{y}{result} \end{bmatrix}$$

The length of the direction vector can be calculated using the **Pythagorean theorem**. If the vector is (x, y), the length is:

$$Length = \sqrt{x^2 + y^2}$$

Then, the normalized vector is obtained by dividing each component of the direction vector by the length.

3. Updating the Position

Now that we have a normalized direction vector, we can update the enemy's position. By adding the normalized vector to the current position, we move the enemy by a small, consistent step toward the target.

- **Formula**:

$$NewPos = CurrentPos + NormalizedDirect \times Speed$$

Here, the **speed** is a scalar value that controls how fast the enemy moves. The normalized direction ensures that the enemy always moves the same distance per frame, no matter where the target is.

Example Code

Here's an example of how this could be implemented in Ruby using the `Vector` class from the **Gosu** library:

```ruby
class Enemy
  attr_accessor :position, :target, :speed

  def initialize(start_position, target_position, speed)
    @position = Gosu::Vector2.new(start_position[0],
start_position[1])
    @target = Gosu::Vector2.new(target_position[0],
target_position[1])
    @speed = speed
  end

  def move
    direction = @target - @position  # Step 1: Calculate
the direction
    direction_length = direction.length
    direction.normalize!  # Step 2: Normalize the vector

    # Step 3: Move the enemy in the direction of the target
    @position += direction * @speed
  end
end
```

Explanation of the Code

- **Initialization**: The `Enemy` class takes a starting position, a target position, and a speed. The positions are stored as `Gosu::Vector2` objects.
- **Moving the Enemy**:
  - The direction is computed by subtracting the current position from the target position (`@target - @position`).
  - The direction is then normalized using `normalize!`, which adjusts its length to 1.

- The enemy moves by adding the normalized direction, scaled by the speed, to the current position (`@position += direction * @speed`).

Benefits of Vector-Based Movement:

- **Simplicity**: The use of vector operations simplifies the process of calculating movement, as vectors inherently handle both direction and magnitude.
- **Precision**: By normalizing the direction, we ensure that the enemy moves at a constant speed regardless of the distance to the target.
- **Flexibility**: Vectors allow easy adjustments for features like diagonal movement or varying speeds, simply by modifying the direction vector or speed.

This approach provides smooth and efficient movement for enemies, ensuring they always move in a straight line toward their target with consistent speed.

Health Bar

A **Health Bar** is an essential part of any game, providing players with a visual representation of a character’s or enemy's remaining health. In this game, the health bar is implemented using two rectangles: one for **Max Health** and another for **Current Health**. These two rectangles allow players to easily observe how much health is left and how close they are to losing all their health.

How Does It Work?

The health bar is composed of two key parts:

1. **Max Health**: This represents the full capacity of health and is visualized as a background bar.

2. **Current Health**: This shows how much health remains and is drawn as a foreground bar inside the max health bar. Its length dynamically changes based on the character's current health.

To calculate the length of the **Current Health Bar**, we use the following formula:

$$\text{currentHealthBarLength} = \frac{\text{currentHealth}}{\text{maxHealth}} \times \text{barLength}$$

This formula ensures that the current health bar accurately reflects the player's health as a percentage of their maximum health. The length of the current health bar is proportional to how much health is left.

Step-by-Step Breakdown:

1. **Max Health Bar**: The max health bar will always have a fixed length that represents the full health capacity.
2. **Current Health Bar**: The current health bar's length changes dynamically depending on the player or enemy's current health. This is computed using the formula above, which scales the current health relative to the max health.

Example Code

In the following example, we use a simple implementation to create a health bar:

```
class HealthBar
  attr_accessor :current_health, :max_health, :bar_length

  def initialize(max_health, bar_length)
    @max_health = max_health
    @current_health = max_health  # Initially, health is
full
    @bar_length = bar_length
  end
```

```
  def draw
    # Draw the background bar (Max Health)
    draw_rectangle(0, 0, @bar_length, 20,
Gosu::Color::WHITE)

    # Calculate the current health bar length based on
current health
    current_health_length = (@current_health.to_f /
@max_health) * @bar_length

    # Draw the current health bar
    draw_rectangle(0, 0, current_health_length, 20,
Gosu::Color::RED)
  end

  def draw_rectangle(x, y, width, height, color)
    # This is a placeholder for the actual drawing code,
using Gosu library
    # Replace this with actual drawing logic as needed.
    Gosu.draw_rect(x, y, width, height, color)
  end
end
```

Explanation:

- `initialize` **Method**: This method sets the maximum health (`max_health`), the current health (`current_health`), and the length of the health bar (`bar_length`). Initially, the current health is set to be equal to the max health.
- `draw` **Method**:
  - First, the max health bar is drawn as a white rectangle.
  - Then, the length of the current health bar is calculated using the formula and drawn as a red rectangle. The width of this rectangle will dynamically change based on the player's current health.

Why This Works:

- **Dynamic Update**: The length of the current health bar changes as the `current_health` value changes. This provides a real-time visual update to the player about their health status.
- **Proportional Representation**: The health bar is proportional to the player's health. If the player's health is half, the current health bar will be half the length of the max health bar.

This approach ensures that the health bar is easy to implement and provides an intuitive visual representation of health in the game.

Enemy Direction Facing, Archer Facing, and Arrow Orientation

In many games, the orientation of characters, enemies, and projectiles is crucial for gameplay dynamics. For this game, I use a mathematical approach to calculate the **direction facing** for both the enemy and the tower's archer, as well as the **orientation of arrows**.

1. Enemy Direction Facing

The enemy’s direction facing is determined by calculating the **angle of movement** relative to the target. The direction the enemy is facing will change depending on its position and the target’s location.

2. Tower's Archer Facing Direction

Similarly, the **archer’s facing direction** in the tower is determined using the same method, but instead of using the position of the enemy, we calculate the direction based on the position of the **target** relative to the **archer**’s position (usually the tower).

3. Arrow Orientation

When the **archer** shoots an **arrow**, the arrow must face in the direction it was shot. To set the correct **orientation** of the arrow, the angle calculated using the `atan2` function is used.

The arrow's **initial orientation** is determined by the direction vector between the **archer's position** and the **target position**. This angle ensures that the arrow will travel in the correct direction after being fired.

How to Determine the Facing Direction

To determine the direction an enemy is facing, follow these steps:

1. **Find the Direction Vector:** Calculate the direction vector using the difference in coordinates between the **current position** and the **target position**. The vector will have an x and y component.
2. **Calculate the Angle Using the Tangent Function:** The direction angle is calculated using the **tangent inverse** function, `Math.atan2(x, y)`. This function returns the angle of the vector in radians. It handles the entire unit circle and accounts for all four quadrants, providing a more accurate angle than the basic `Math.atan` function.
3. **Determine the Facing Direction:** Based on the angle value returned from `Math.atan2`, the facing direction is determined by comparing the angle to the four key quadrants:
    - **Right**: The direction is facing to the right if the angle is between `−π/4` and `π/4` (roughly -0.78 to 0.78 radians).
    - **Up**: The direction is facing up if the angle is between `π/4` and `3π/4` (roughly 0.78 to 2.35 radians).
    - **Left**: The direction is facing left if the angle is greater than `3π/4` or less than `−3π/4` (roughly ±2.35 radians).
    - **Down**: The direction is facing down if the angle is between `−3π/4` and `−π/4` (roughly -2.35 to -0.78 radians).

Example Code

```ruby
def calculate_enemy_facing_direction(x, y, target_x,
target_y)
  # Find the direction vector
  direction_x = target_x - x
  direction_y = target_y - y

  # Calculate the angle of the direction using atan2
  angle = Math.atan2(direction_y, direction_x)

  # Determine the facing direction based on the angle
  case angle
  when -Math::PI / 4..Math::PI / 4
    "Right"
  when Math::PI / 4..3 * Math::PI / 4
    "Up"
  when -Math::PI..-3 * Math::PI / 4
    "Down"
  else
    "Left"
  end
end
```

How It Works:

- **Direction Calculations**: By calculating the angle between the positions, we can determine which quadrant the vector falls into and thereby determine the facing direction.
- **Arrow Orientation**: The angle returned by `Math.atan2` is then applied to the arrow's sprite or graphics to ensure that the arrow faces the right direction when launched.

Why Use `Math.atan2`?

The function `Math.atan2(y, x)` is used instead of `Math.atan(y/x)` because `atan2` returns the correct angle across all four quadrants, ensuring that the calculations work in any direction (e.g., upwards, downwards, etc.). This is especially useful for determining the facing

direction of the enemy or archer, as it accounts for both the x and y values simultaneously.

Enemy Delay and Spawn Interval

In many games, especially in wave-based or timed spawning systems, it is important to control how frequently enemies spawn. This helps to create balanced gameplay by controlling the flow of enemies in the game world. One common method to implement this is by introducing a **delay** or **interval** between enemy spawns.

In this section, we'll explain how the **enemy spawn delay** works, how it's calculated, and how it is implemented in the game using a simple timer-based system.

1. Spawn Delay Concept

The core idea behind spawn delay is to ensure that enemies do not spawn immediately one after another but instead, after a set interval of time. This is controlled by the time elapsed since the last enemy was spawned. The formula for this interval is:

Gosu.milliseconds−@lastSpawn≥spawnDelay\text{Gosu.milliseconds} - @lastSpawn \geq \text{spawnDelay}

Here:

- `Gosu.milliseconds` gives the current time in milliseconds since the game started.
- `@lastSpawn` is a variable that holds the time (in milliseconds) when the last enemy was spawned.
- `spawnDelay` is a constant (or variable) that specifies the minimum amount of time that must pass before a new enemy can spawn.

2. How It Works

- **Initial Setup**: When the game starts or when the spawn system is initialized, we set the `@lastSpawn` variable to the current time (`Gosu.milliseconds`). This marks the starting point of the spawn system.
- **Checking the Condition**: During each frame of the game, we check if the current time minus `@lastSpawn` is greater than or equal to `spawnDelay`. If this condition is true, it means enough time has passed since the last spawn, and a new enemy can be spawned.
- **Spawning an Enemy**: If the condition is met, we spawn a new enemy and then update `@lastSpawn` to the current time (`Gosu.milliseconds`), essentially resetting the timer for the next spawn.

3. Implementing Spawn Delay

Here's an example of how this could be implemented in Ruby using Gosu, a popular game development library for Ruby:

Example Code

```
class EnemySpawner
  # Set the spawn delay (in milliseconds)
  SPAWN_DELAY = 2000 # 2 seconds

  def initialize
    # Store the time of the last spawn
    @last_spawn = Gosu.milliseconds
  end

  def update
    # Check if the spawn interval has passed
    if Gosu.milliseconds - @last_spawn >= SPAWN_DELAY
      spawn_enemy
      @last_spawn = Gosu.milliseconds  # Reset the spawn
time
    end
  end
```

```
  def spawn_enemy
    # Logic to spawn an enemy (e.g., create a new enemy
object)
    puts "Enemy spawned!"
  end
end
```

4. Explanation of Code

1. `SPAWN_DELAY`: This is the time interval between each spawn, set to 2000 milliseconds (or 2 seconds) in this example.
2. `@last_spawn`: This variable stores the timestamp of the last spawn, initialized to the current time when the spawner is created.
3. `update` **Method**: This method is called every frame to check if the time difference between the current time and the last spawn time is greater than or equal to `SPAWN_DELAY`. If it is, it calls the `spawn_enemy` method and resets `@last_spawn` to the current time.
4. `spawn_enemy` **Method**: This is a placeholder for the actual logic to spawn an enemy, such as creating a new enemy object and adding it to the game world.

5. Why Use This Delay System?

- **Prevents Overloading**: Without a spawn delay, enemies might flood the game world too quickly, overwhelming the player. Introducing a delay ensures that enemies spawn at a controlled rate.
- **Game Balance**: Adjusting the delay between spawns allows for difficulty scaling, where you can make enemies spawn faster over time or slower based on the player's progress.
- **Performance**: A fixed spawn interval reduces the need to check or handle multiple enemy spawns every frame, improving game performance and reducing unnecessary computations.

Detecting Enemies Inside a Tower's Area

In games, it’s common for towers or other defensive structures to have an area of influence within which they can interact with enemies, whether by targeting, attacking, or performing other actions. To implement this, we need a method to detect if an enemy is inside the tower's area of influence.

This section explains how to detect when an enemy enters a tower's range and how to mark that enemy as the tower's target.

1. Detecting the Enemy Inside the Tower's Area

To check if an enemy is within the tower’s range, we need to calculate the **distance** between the tower and the enemy. If this distance is less than or equal to the **tower’s radius**, then the enemy is considered inside the tower’s area.

The distance can be calculated using the **distance formula** for two points (in this case, the tower and the enemy). The formula is:

$$\text{distance} = \sqrt{(enemy_x - tower_x)^2 + (enemy_y - tower_y)^2}$$

Where:

- enemyxenemy_x and enemyyenemy_y are the coordinates of the enemy.
- towerxtower_x and towerytower_y are the coordinates of the tower.
- The result of this formula gives the **straight-line distance** between the two points.

If the calculated **distance** is less than or equal to the tower’s **radius**, then the enemy is within range of the tower.

2. How to Mark the Enemy

Once we determine that an enemy is inside the tower's area, the next step is to **mark the enemy** so that the tower can target it. This involves setting the tower's **target variable** to the enemy.

The process works as follows:

- The tower checks if any enemy is within its radius.
- If an enemy is within range, the tower's **target variable** is updated to reference that enemy. This marks the enemy as the current target of the tower.

Here's a breakdown of how this can be executed:

3. Code Example

```
class Tower
  attr_accessor :target

  # Define the tower's position and radius
  def initialize(x, y, radius)
    @x = x
    @y = y
    @radius = radius
    @target = nil  # No target initially
  end

  # Check if an enemy is inside the tower's range
  def detect_enemy(enemy)
    # Calculate the distance between the tower and the
enemy
    distance = Math.sqrt((enemy.x - @x)**2 + (enemy.y -
@y)**2)

    # If the enemy is within the tower's radius, mark it as
the target
    if distance <= @radius
      @target = enemy
    end
```

```
  end
end

class Enemy
  attr_accessor :x, :y

  def initialize(x, y)
    @x = x
    @y = y
  end
end
```

4. Explanation of the Code

1. **Tower Class**:
   - The `Tower` class defines the tower's position (`@x`, `@y`) and its **radius** (`@radius`), as well as the **target** (`@target`), which initially is `nil`.
   - The method `detect_enemy` calculates the distance between the tower and an enemy using the distance formula.
   - If the distance is less than or equal to the tower's radius, it marks the enemy by assigning the `enemy` to the tower's `@target` variable.
2. **Enemy Class**:
   - The `Enemy` class contains the position (`@x`, `@y`) of the enemy.
3. **Distance Calculation**:
   - The distance between the enemy and the tower is calculated using the Pythagorean theorem (`Math.sqrt((enemy_x - tower_x)^2 + (enemy_y - tower_y)^2)`).
   - If the distance is within the tower's radius, the `@target` of the tower is set to that enemy.

5. Summary

- **Detecting the Enemy**: By calculating the straight-line distance between the enemy and the tower, we can easily determine if the

enemy is inside the tower’s range. If the distance is less than or equal to the tower's radius, the enemy is considered to be within the tower's area.
- **Marking the Enemy**: Once an enemy is inside the tower's area, the tower marks that enemy by updating its target variable. This allows the tower to focus on the enemy for actions like attacking or defending.

Collision Detection for Arrows and Enemies

In the game, collision detection ensures that interactions, like an arrow hitting an enemy, happen at the right moment. This section explains the mechanics of detecting collisions between arrows and enemies, how arrows are created, and how the archer’s attack sequence is executed.

1. Overview of the Process

- The **tower’s archer** creates an `Arrow` instance when an enemy is detected within range.
- The arrow is projected toward the enemy, guided by its **position** and **direction**.
- When the arrow reaches the enemy’s position, a **collision** is detected.
- On collision:
  - The arrow is destroyed.
  - The enemy takes damage.
  - The archer performs an **attack animation**.

2. Execution Steps

1. **Arrow Creation**
   - When the tower detects an enemy within range, it creates an arrow and initializes its position and direction based on the

tower's location and the enemy's position.

2. **Arrow Movement**
   - The arrow moves toward the enemy's position by updating its coordinates based on its direction and speed.
3. **Collision Detection**
   - Check the distance between the arrow and the enemy.
   - If the distance is less than or equal to a small threshold (e.g., the arrow's size or hitbox), the arrow is considered to have hit the enemy.
4. **Arrow Destruction and Attack Animation**
   - Upon collision, the arrow is removed from the game.
   - The archer starts the attack animation.

3. Code Example

```
class Arrow
  attr_accessor :x, :y, :direction, :speed

  def initialize(x, y, direction, speed)
    @x = x
    @y = y
    @direction = direction  # [dx, dy] as normalized vector
    @speed = speed
  end

  # Update arrow position
  def move
    @x += @direction[0] * @speed
    @y += @direction[1] * @speed
  end
end

class Archer
  attr_accessor :position, :animation_state

  def initialize(x, y)
```

```ruby
    @position = [x, y]
    @animation_state = :idle  # :idle, :attacking
  end

  # Trigger attack animation
  def attack
    @animation_state = :attacking
  end

  # Reset to idle state after attack
  def reset
    @animation_state = :idle
  end
end

class Enemy
  attr_accessor :x, :y, :health

  def initialize(x, y, health)
    @x = x
    @y = y
    @health = health
  end

  # Reduce health when hit
  def take_damage(amount)
    @health -= amount
  end
end

# Collision detection
def detect_collision(arrow, enemy)
  distance = Math.sqrt((arrow.x - enemy.x)**2 + (arrow.y -
enemy.y)**2)
  return distance <= 5  # Assume 5 is the collision
threshold
end

# Game loop example
tower_archer = Archer.new(100, 100)
```

```
enemy = Enemy.new(200, 200, 100)
arrow = Arrow.new(tower_archer.position[0],
tower_archer.position[1], [1, 1], 5)

# Simulate game loop
loop do
  arrow.move

  if detect_collision(arrow, enemy)
    arrow = nil  # Destroy arrow
    enemy.take_damage(20)
    tower_archer.attack  # Start animation
    break
  end
end
```

4. Explanation of Concepts

- **Arrow Creation**: The `Arrow` class initializes the arrow's position, direction, and speed. The direction is a normalized vector pointing toward the target enemy.
- **Arrow Movement**: The `Arrow#move` method updates the arrow's position based on its speed and direction.
- **Collision Detection**: The distance between the arrow and the enemy is calculated using the distance formula. If the distance is within a small threshold, a collision is detected.
- **Arrow Destruction**: The arrow is removed (`arrow = nil`), representing its destruction upon hitting the enemy.
- **Archer Animation**: The `Archer#attack` method triggers the attack animation. The animation ends or resets after a specific duration.

5. Execution Flow

1. **Tower Targets Enemy**
   When an enemy enters the tower's range, an arrow is created and launched toward the enemy.

2. **Arrow Moves Toward Enemy**
   The arrow continuously updates its position based on its direction.
3. **Collision Check**
   Each frame, the game checks whether the arrow has collided with the enemy using the distance formula.
4. **On Collision**
   - The arrow is destroyed.
   - The enemy's health is reduced.
   - The archer starts the attack animation.
5. **Animation Reset**
   After the attack animation ends, the archer returns to the idle state.

6. Summary

Collision detection for arrows and enemies involves:

- Calculating distances to check for overlap.
- Destroying the arrow and applying damage on collision.
- Triggering the archer's attack animation for visual feedback.

This process ensures smooth interaction between game elements and creates an immersive experience for players.

Testing

To ensure the game functions correctly and all components work as intended, I implemented **unit testing** for each module. Unit testing allows us to isolate individual parts of the game, verify their behavior, and quickly identify and fix issues.

Testing Approach

For the game, I tested the following components:

1. **Enemy Movement**
   Verified the correctness of direction calculation, vector normalization, and position updates.
2. **HP Bar**
   Tested the formula for current health bar scaling based on maximum health and ensured the visuals matched expected values.
3. **Collision Detection**
   Confirmed that arrows correctly detect collisions with enemies using distance calculations.
4. **Tower Targeting**
   Checked that the tower correctly detects enemies within its range and prioritizes the nearest target.
5. **Animation Triggers**
   Validated that animations (e.g., attack, idle) are triggered and reset as expected.

Conclusion and Future Work

1. Summary of Achievements

During this project, I successfully developed a functional game framework incorporating key mechanics such as enemy movement, tower targeting, health bar updates, collision detection, and animation triggers. Here are the main achievements:

- **Enemy Movement**: Implemented precise vector-based movement with direction calculation and speed control.
- **Tower Mechanics**: Designed a robust targeting system where towers detect and attack enemies within range.
- **Collision Detection**: Built accurate arrow-enemy collision logic, ensuring smooth gameplay interaction.
- **Animations and Visuals**: Integrated animations for dynamic

feedback, including archer attacks and enemy behaviors.
- **Scalable Health Bar**: Created a responsive health bar system tied to enemy health for real-time updates.
- **Testing**: Applied unit tests to verify each module's functionality, ensuring a stable and bug-free game.

These features contribute to an engaging and interactive game experience, showcasing the use of Gosu and mathematical principles like vector operations and trigonometry.

2. Possible Improvements and Future Features

While the game is functional, several areas can be enhanced or expanded to improve gameplay and player experience:

1. **Enhanced AI for Enemies**
    - Introduce varied enemy behaviors, such as evasion, group formation, or pathfinding.
    - Add different enemy types with unique abilities and strategies.
2. **Advanced Tower Mechanics**
    - Implement multiple tower types (e.g., splash damage, slowing, long-range).
    - Allow upgrades for towers, improving range, speed, or damage.
3. **Dynamic Level Design**
    - Add procedurally generated levels or maps to keep gameplay fresh.
    - Include obstacles or terrain affecting movement and targeting.
4. **Visual and Sound Enhancements**
    - Improve graphics with better sprite designs and animations.
    - Add immersive sound effects and background music.
5. **Game Progression System**
    - Introduce a scoring or leveling system to provide players with goals.
    - Add new waves of enemies with increasing difficulty.

6. **Multiplayer Mode**
    - Allow cooperative or competitive multiplayer gameplay.
7. **Optimization and Portability**
    - Optimize the game for smoother performance on low-end devices.
    - Explore cross-platform compatibility for wider reach.

This project demonstrates the successful implementation of a tower defense game using Gosu, emphasizing modularity and maintainability. By incorporating future improvements, the game can evolve into a more complex and engaging experience for players. The foundation laid here opens the door to countless creative possibilities in game development.

References

This project utilized various resources, tools, and frameworks to streamline development and ensure quality implementation. Below is a list of key references:

1. Frameworks and Libraries

- **Gosu**
    - Primary library for graphics, input, and audio in Ruby.
    - Documentation: https://www.libgosu.org/
- **Ruby Standard Library**
    - Utilized for mathematical operations (`Math.atan2`), timing (`Time`), and other core functionalities.
    - Documentation: https://ruby-doc.org/

---

2. Tutorials and Guides

- **Gosu Tutorial: Creating a 2D Game in Ruby**
  - Offered a foundational understanding of game creation with Gosu.
  - Tutorial Link: https://www.ruby-gosu-tutorial.com/
- **Vector Math in Game Development**
  - 6.2 Vector Addition & Subtraction (full lesson) | grade 12 MCV4U | jensenmath.ca
  - Vector Normalization EXPLAINED - Game Dev Math

---

3. Additional Resources

- **Stack Overflow**
  Referenced for troubleshooting specific implementation issues.
  - Website: https://stackoverflow.com/
- **Community Forums**
  Gosu community forums provided tips and best practices for optimizing game performance.
  - Forum Link: https://www.libgosu.org/community.html
