Tiny Swords

2D Platformer in Unity

Welcome to "Tiny Swords," a 2D action-adventure RPG developed in Unity! In this game, players step into the boots of a valiant knight exploring a mystical world filled with danger, treasure, and secrets. Drawing inspiration from classic RPGs, this project combines **fast-paced combat**, **inventory management**, **enemy AI**, and an **interactive UI** to create an immersive experience.

Team: Sanika Narmitwar (B22CS046), Vindhya Jain (B22AI060)

View Gameplay Video

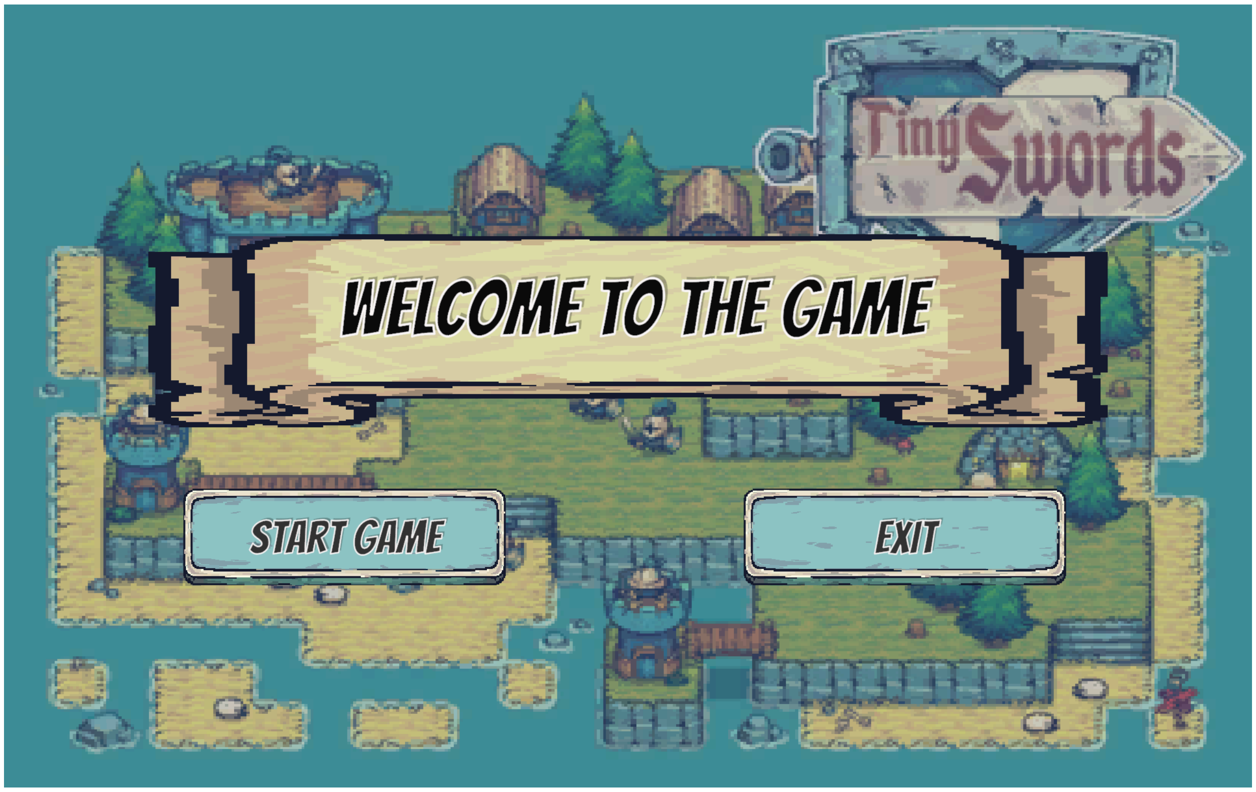


Table of Contents

Introduction

Game Overview

The game is set in a handcrafted fantasy world, built using the Tiny Swords asset pack, which gives it a charming pixel-art aesthetic. As the knight, players must battle enemies, gather resources, and manage their inventory to survive.

The game blends real-time combat, inventory management, and exploration in a pixel-art-styled environment created using the Tiny Swords asset pack.

Key gameplay elements include:

- Character control (movement, attacking, interacting with objects)
- Enemy AI that pursues and attacks the player
- A dynamic inventory system with consumable items
- A progression system (EXP, leveling up)
- Interactive UI (menus, health bars, inventory slots)

This project was designed using Unity's 2D development pipeline, including sprite animation, physics-based combat, C# scripting, and UI programming.

Development Tools

The game was built using:

1. Unity Engine 6 – For core game development
2. C# – For scripting player mechanics, AI, and UI
3. Tiny Swords Asset Pack – For pixel-art visuals and tileset
4. Unity's Cinemachine – For smooth camera follow
5. Unity's Tilemap System – For world-building
6. Unity's Animator - For player and enemy movements, water

Movement and Controls

The player character features smooth 8-directional movement using Unity's Rigidbody2D physics for realistic acceleration and deceleration. Key controls include:

WASD – Move the character
k – Sword attack
Left Click - Use Item
Right Click - Drop Item
Esc - Toggle Stats Manager

World Design and Layer-Based Collision Handling

For our game, we have designed a layered 2D environment using the **Tiny Swords asset pack**, which features stylized pixel-art elements such as trees, houses, mountains, and various terrain features. The world is structured to provide both visual depth and meaningful gameplay constraints through a custom **layer-based system**.

Layer-Based World Structure

The game environment is organized into multiple layers to control player interactions with different elements in the world. Each layer defines whether the player can move onto a surface or if the object serves as a visual obstruction without collision:

- **Collision Low**: Elements on this layer block player movement at a lower elevation level (e.g., fences, rocks, walls at ground level).
- **Collision High**: Blocks player movement at higher elevations, typically used for objects that are visually taller or inaccessible without elevation changes (e.g., mountain tops).
- **Non-Collision Low**: Includes visuals such as grass, shadows, or objects the player can walk over or under without restriction.
- **Non-Collision High**: Used for tall visual-only elements like tree canopies or rooftops that appear above the player but do not impede movement.

This layer separation is essential for implementing depth and verticality in a 2D plane while maintaining intuitive player interactions.

Environmental Elements and Interactions

Key objects such as **trees, houses, and mountains** are placed strategically in the world, with specific interactions:

- **Trees**: The player can walk behind the tree trunks, creating a sense of depth, but cannot walk through them due to collision detection on the trunk base.
- **Houses**: Similar to trees, houses use layered rendering and collision to ensure the player walks around them realistically.
- **Mountains**: Mountains are designed as multi-layered obstacles. Players cannot ascend the mountain directly. Instead, access is controlled through **staircase objects**, which act as navigational waypoints allowing vertical movement from one collision layer to another.

All layers and interactions are implemented within **Unity**, using sorting layers and colliders to enforce movement rules. Sprite renderers and Z-indexing ensure correct visual stacking, while polygon colliders and rigidbody components manage physical interactions. Layer masks are used to define what the player can or cannot collide with based on their current position and context.

Player Character Design and Interaction System

The player character in our game is designed to be fully interactive and responsive, both visually and mechanically. The character plays a central role in navigating the world, collecting resources, engaging in combat, and progressing through levels.

Player Animation

The player is animated using a set of **2D sprite animations** that cover key actions:

- **Idle Animation**: Played when the player is stationary, giving a sense of life and presence even when inactive.

- **Walking Animation**: Triggered while the player moves, creating fluid directional movement across the game world.

- **Sword Attack Animation**: Activated during combat interactions, synchronized with hit detection to register damage to enemies.

All animations are managed using Unity's **Animator** component with transitions controlled by player input and game states.

Player Interactions

The player is designed to interact with several types of elements in the game:

- **World Interactions**: The player's ability to move, collide, or pass behind objects is governed by the world's layer system. This allows for rich environmental interaction, such as walking behind trees but not through them, or climbing mountains only via staircases.

- **Collectible Items**: Throughout the world, players can pick up various items, such as healing foods or speed boosts. These are stored in the **inventory system** and can be used when needed to affect gameplay.

- **Enemies and Combat**: The player engages enemies through sword attacks, initiating combat animations and dealing damage upon successful hits. Enemies can also damage the player, leading to a reduction in health

Health and Experience System

The player has a **health system** with a maximum of **20 HP**. Health can be reduced by enemy attacks and restored using consumable healing items. In addition to health, the player has an **experience (EXP) bar** that increases with successful actions like defeating enemies. The EXP bar provides **visual feedback** to the player and plays a role in level progression and ability enhancements.

Consumable Effects

The player can use items that provide various effects:

- **Healing Items**: Restore lost health.
- **Speed Boosts**: Temporarily increase movement speed, allowing faster navigation or evasive maneuvers.

These items are managed through an inventory interface and activated via player input.

Unity Implementation

The player GameObject in Unity includes the following components:

- **Sprite Renderer**: Displays the player's current sprite based on animation state.
- **Capsule Collider 2D**: Handles collision detection with the environment, enemies, and items.
- **Rigidbody 2D**: Allows physics-based movement and interaction.
- **Animator**: Controls and transitions between animations based on player actions and states.
- **Custom Scripts**: Manage input handling, movement, combat logic, health/EXP tracking, item usage, and inventory interactions.

Enemy Design and Combat Behavior

The enemies in our game serve as active obstacles that challenge the player through strategic combat and responsive behavior. Their design incorporates animation, movement logic, aggro detection, and combat mechanics that are consistent with the player's interaction rules.

Enemy Animation and Movement

Enemies are animated using 2D sprite animations that correspond to different behavioral states:

- **Idle Animation**: Played when the enemy is not engaging the player.

- **Chasing Animation**: Activated when the player enters the enemy's detection radius, indicating pursuit.
- **Attack Animation**: Triggered when the enemy is in range and performs a melee attack on the player.

Enemies move around the game world using pathfinding and physics-based movement. They are bound by the same **world collision system** as the player, ensuring consistent interaction with the environment (e.g., cannot walk through trees or climb mountains without designated paths).

Aggro and Detection System

Each enemy has a **circular detection zone** implemented via a **Circle Collider 2D**, which serves as its **aggro range**:

- If the player enters this zone, the enemy becomes alerted and begins **chasing the player**.
- The **aggro zone is biased toward the front of the enemy**, making it less likely for enemies to detect the player from behind. This encourages strategic movement and stealth.

Combat Mechanics

When the player and enemy engage in combat, the following rules apply:

- **Chase and Attack**: Once within attack range, the enemy transitions into an attacking state and deals damage to the player.
- **Knockback Effect**: When either the player or enemy takes damage, a **knockback force** is applied, pushing the affected character back slightly. This creates space and makes combat feel more dynamic.
- **Attack Cooldown**: After an attack, both the player and the enemy experience a **short cooldown period** during which they cannot attack again. This encourages the player to use movement and timing to evade or reposition.

Health and Death Conditions

Both player and enemies have health systems:

- If an enemy’s health reaches zero, it **dies** and is removed from the game.
- If the player's health reaches zero, a **death event** is triggered (e.g., game over or respawn).
- The player can use **food items** to heal or **speed buffs** to enhance evasion and mobility during combat.

Unity Implementation

Each enemy GameObject in Unity contains the following components:

- **Sprite Renderer**: Displays the current animation frame.
- **Rigidbody 2D**: Enables movement and physical interaction.
- **Capsule Collider 2D**: Used for general collision with the world and player.
- **Circle Collider 2D**: Acts as the aggro/detection range.
- **Custom Scripts**: Handle AI logic for idle, chase, and attack behavior, health tracking, damage calculation, knockback effects, and cooldown timing.

User Interface

The user interface (UI) for the game was designed and implemented using Unity's built-in UI system, focusing on clarity, accessibility, and immersive interaction. The interface is divided into several canvases and elements, each serving a specific purpose and activated during different stages of the gameplay. The following are the key UI components developed:

Main Menu (MainMenuCanvas)

- The first interface screen shown when the game launches.
- Includes buttons for Start and Exit functionalities.
- Implemented using Unity's Canvas, Button, and Text components.
- The main menu buttons are linked to a custom script that contains a LoadScene(string sceneName) function, which uses SceneManager.LoadScene to transition between scenes. This function was assigned to the OnClick() events of the UI buttons directly through the Unity Inspector, allowing seamless scene transitions without hardcoding specific button actions.

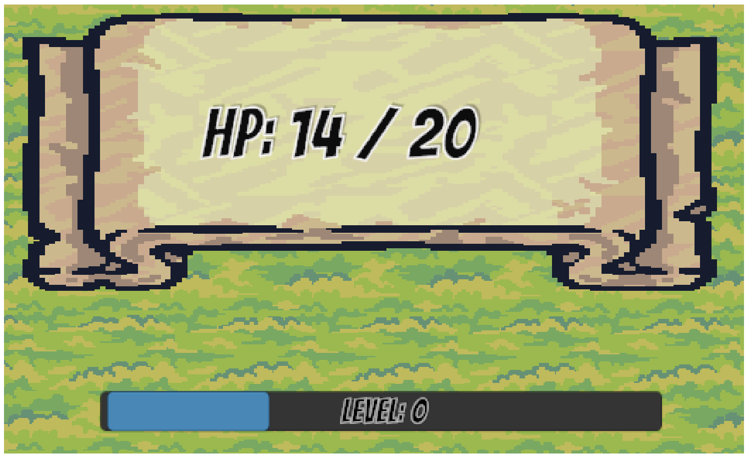
Health UI

- Permanently visible during gameplay.
- Displays the Current Health and Maximum Health of the player.
- Updated in real-time through script references to the player's health stats

Experience and Level Display (EXPCanvas)

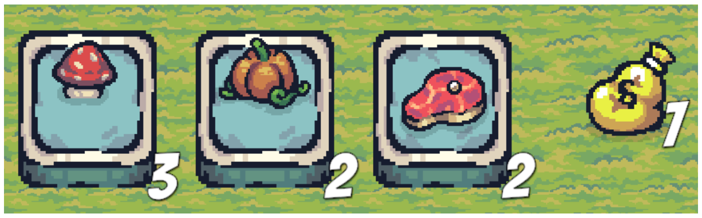
- A scroll-based UI element that tracks the player's Level.
- Level increases as the player gains Experience (XP) by killing enemies

- Experience gain is dynamically visualized through a progress bar or scroll fill.
- Managed via scripting to update XP in real-time and trigger level-up effects.



Inventory System (InventoryCanvas)

- Displays collected items in a grid-based layout using Unity’s GridLayoutGroup.
- Features 3 inventory slots, each supporting a stack size of up to 10 items.
- Item data is managed through a custom Inventory script that handles stack size, slot management, and interaction.
- Left-clicking on an item invokes a "Use" action defined per item type (e.g., healing, speed boost).
- Right-clicking on an item triggers a "Drop" action, removing it from the inventory and spawning it into the game world.
- The Gold Count is displayed on the inventory screen, showing the player's currently held gold amount.
- Uses Unity’s Image, Button, and Text components for layout and interactivity.

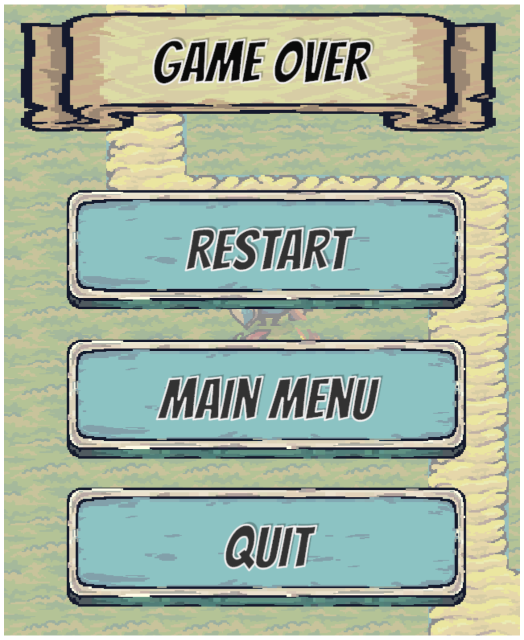


Stats Display (StatsCanvas)

- Toggled using the Escape key
- Shows detailed attributes such as speed, damage, etc.
- These stats are updated in real-time and displayed using Text UI elements.

Game Over Screen (ExitCanvas)

- Triggered upon player death.
- A new UI screen overlays the gameplay and offers three options:
    - Restart: Restarts the game.
    - Main Menu: Returns the player to the main menu screen.
    - Quit: Exits the application.
- Implemented using buttons tied to scene management scripts using Unity’s SceneManager.


GAME OVER
RESTART
MAIN MENU
QUIT