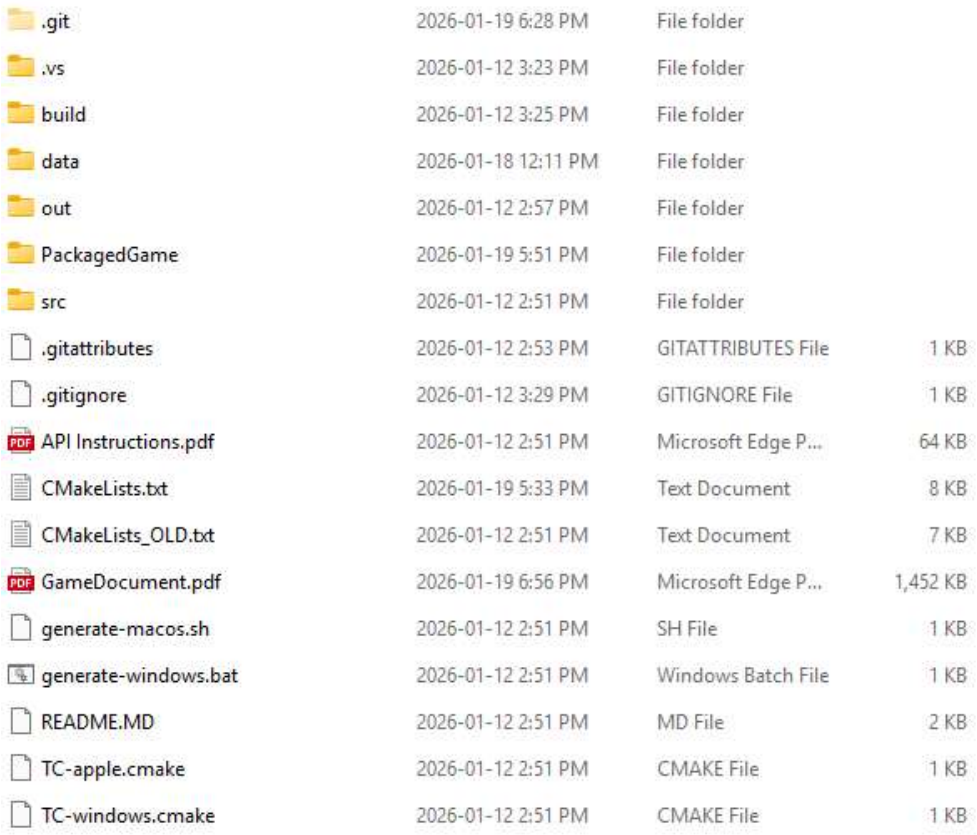
Bug Swarm: Tactical Eradication Protocol
Please Note: that Zombies refers to Bugs/ Enemies changes were made during development.

The Game.exe located at PackagedGame

\



.git	2026-01-19 6:28 PM	File folder	
.vs	2026-01-12 3:23 PM	File folder	
build	2026-01-12 3:25 PM	File folder	
data	2026-01-18 12:11 PM	File folder	
out	2026-01-12 2:57 PM	File folder	
PackagedGame	2026-01-19 5:51 PM	File folder	
src	2026-01-12 2:51 PM	File folder	
.gitattributes	2026-01-12 2:53 PM	GITATTRIBUTES File	1 KB
.gitignore	2026-01-12 3:29 PM	GITIGNORE File	1 KB
API Instructions.pdf	2026-01-12 2:51 PM	Microsoft Edge P...	64 KB
CMakeLists.txt	2026-01-19 5:33 PM	Text Document	8 KB
CMakeLists_OLD.txt	2026-01-12 2:51 PM	Text Document	7 KB
GameDocument.pdf	2026-01-19 6:56 PM	Microsoft Edge P...	1,452 KB
generate-macos.sh	2026-01-12 2:51 PM	SH File	1 KB
generate-windows.bat	2026-01-12 2:51 PM	Windows Batch File	1 KB
README.MD	2026-01-12 2:51 PM	MD File	2 KB
TC-apple.cmake	2026-01-12 2:51 PM	CMAKE File	1 KB
TC-windows.cmake	2026-01-12 2:51 PM	CMAKE File	1 KB

Game Overview
Bug Swarm: Tactical Eradication Protocol is a top-down action simulation built to stress-test real-time entity management at extreme scale. The game supports approximately 20,000 to 200,000 simultaneous enemies while maintaining responsive controls and stable frame pacing.

Gameplay is designed around emergent swarm pressure rather than scripted encounters. The player must destroy hive spawners while navigating dense enemy fields that continuously re-form and push toward the player.

A core mechanic is player scaling. Player size directly changes movement speed, survivability, and attack radius, creating meaningful tradeoffs: small is fast, fragile and weaker damage output whereas large is slow and durable with larger area control and higher damage output.

The main goal of the project is to demonstrate scalable system design, data-oriented programming, and performance-driven gameplay using only C++ and the provided API.

Core Gameplay Loop

- Hives spawn enemies around the player over time.
- Enemies seek the player, forming dense swarms.
- The player clears space with area attacks and cooldown abilities.
- Player scaling changes how aggressively the player can engage.
- Difficulty ramps by increasing spawn pressure and the active entity cap from the menu screen:

High-Level Architecture

The project is organized into a small set of systems with clear ownership.

ZombieSystem

- Owns enemy data, behavior updates, movement resolution, and entity lifetime.
- Implements SoA storage, LOD scheduling, spatial grid building, and collision-safe movement.

NavGrid

- Provides world bounds, collision queries (circle against obstacles), and flow-field helpers.

Player + Attacks

- Handles player movement, scaling, ability cooldowns, and area damage application.

Renderer / UI

- Draws the world, entities, and debug overlays (optional density view).

The update loop is structured so that simulation work stays in tight passes over contiguous arrays, and rendering is separate from the logic that moves and kills entities.

Data Layout and Memory Strategy

Enemies are stored using Structure-of-Arrays (SoA). Each field is a contiguous array sized to maxCount, such as:

- posX[], posY[]
- velX[], velY[]
- hp[]
- type[], state[]
- fearTimerMs[], attackCooldownMs[], flowAssistMs[]

Why SoA matters here

• Update loops touch the same fields for thousands of entities, so contiguous memory reduces cache misses.
• Systems can read only what they need (positions for grid, timers for cooldowns) without pulling extra unused data.
• Removal uses swap-remove, keeping arrays dense and iteration fast.

Entity lifetime avoids runtime allocations: all arrays are allocated once in Init(maxZombies), then reused.

ZombieSystem Technical Deep Dive (Hardest Part)

ZombieSystem is the performance-critical component and includes multiple techniques to keep behavior believable at 200k entities.

Spawning and Initialization

Init(maxZombies, nav) allocates all SoA arrays to maxCount and caches world bounds from NavGrid. It also configures the spatial grid dimensions based on world size and cell size.

Spawn and SpawnAtWorld insert new zombies by writing directly into SoA arrays at index aliveCount, then incrementing aliveCount. Type is chosen via a weighted roll (GREEN, RED, BLUE, PURPLE_ELITE), each with its own stats (speed, HP, touch damage, fear radius, cooldown).

Constant-Time Removal (Swap-Remove)

When a zombie is removed (killed, despawned), the system performs swap-remove:
• Copy last alive zombie data into the removed index
• Decrement aliveCount
• Mark the grid dirty

This ensures removals are O(1) and keeps the SoA arrays packed, which is essential for performance at scale.

Spatial Acceleration: Near-Only Uniform Grid

The most expensive local behavior is separation (avoiding overlap). Doing separation for every zombie against every other zombie is not viable, so the system uses two layers of filtering:

Filter 1: Near list

Only zombies within a radius around the player (sepActiveRadius) are placed into the grid. This is built each time the grid rebuilds.

Filter 2: Uniform grid neighbor queries

The near zombies are inserted into a fixed grid (cellSize = 2 × separation radius). Each zombie

only checks nearby cells (3 × 3 neighborhood), making neighbor access effectively constant time.

Grid build is done with a cache-friendly prefix-sum approach:
- Count zombies per cell (cellCount)
- Prefix-sum into cellStart
- Fill a compact list (cellList) using cellStart as write cursors
- Restore cellStart for query use

This avoids per-cell vectors and keeps iteration linear and contiguous.

Separation Computation (Capped Work)
ComputeSeparation(i) calculates a local push vector that keeps swarms from collapsing into a single point.

Key performance constraints:
- Only runs for near zombies (the ones inserted into the grid).
- Caps neighbor checks (maxChecks = 32) to prevent worst-case spikes in dense clusters.
- Checks own cell first (highest density), then only checks adjacent cells if still under the cap.
- Uses squared distance tests first, and only computes sqrt when within separation radius.

This keeps separation stable, predictable, and scalable.

Collision-Safe Movement: Slide Resolution
Movement uses circle collision checks against NavGrid to avoid clipping into obstacles.
ResolveMoveSlide tries:
- Full move: (x + vx × dt, y + vy × dt)

If blocked:
- X slide: (x + vx × dt, y)

If blocked:
- Y slide: (x, y + vy × dt)

If all fail, the zombie is considered fully blocked for this update. This approach produces stable sliding along obstacles and reduces jitter in tight spaces.

Pop-Out Unstuck System
At very high densities, zombies can spawn inside blocked areas or be pushed into invalid positions. PopOutIfStuck performs a radial search:
- If current position is blocked, scan around in expanding rings
- Test a fixed number of angles per ring
- Pick the first position where IsCircleBlocked is false

This prevents permanent deadlocks without teleporting zombies large distances.

Distance-Based LOD With Frame Staggering

To reach 200k entities, the system reduces work for distant zombies.

Distance tiers:

- Near (within sepActiveRadius): full behavior (seek + optional flow assist + separation + collision move)
- Far (outside sepActiveRadius): reduced update frequency and reduced features
- Very far: even lower frequency

Frame staggering spreads expensive work over multiple frames using a bitmask test on (frameCounter + i). Far zombies often do a cheap drift step using their last velocity, then occasionally recompute velocity and collision.

This avoids large frame-time spikes and keeps the simulation smooth.

Flow Assist Burst (Anti-Stuck Steering)

When zombies are blocked repeatedly or fail to move, the system enables a short flow assist timer. While active, the zombie blends its direct seek direction with NavGrid flow vectors. This helps zombies navigate around obstacles without running full pathfinding per entity.

Damage and Interaction

Zombies apply touch damage when close enough to the player and their attack cooldown is ready. Cooldowns are stored per zombie in attackCooldownMs[] and ticked each frame.

Fear is implemented as a state change plus a timer. Feared zombies flee from the player; if they get far enough away they despawn, reducing entity load and creating a meaningful crowd-control effect.

Performance Summary

Key techniques used to support 20k to 200k entities:

- SoA layout for cache-efficient iteration
- O(1) neighbor queries using a uniform grid (near-only)
- Prefix-sum grid build using contiguous arrays
- Capped separation checks per zombie
- Swap-remove for O(1) deletion
- Distance-based LOD with frame staggering
- Collision-safe movement with axis slide resolution
- Pop-out unstuck search to prevent deadlocks
- Flow assist bursts to improve navigation without heavy computation

Controls and How to Play
• Move: WASD or controller left stick
• Aim: The way your character faces

Attacks (3 abilities, all on cooldown):
Pulse: Blast around you to clear nearby enemies
Slash: A forward swing that hits enemies in front of you and heals you based on how many you kill
Meteor: Drop a big strike a short distance in front of you to wipe a cluster

• Objective: destroy hive spawners and survive escalating waves

Build and Run Instructions

• Generate project files with CMake using the provided API instructions.
• Build using Visual Studio 2022 in Release mode for best performance.
• Run the produced executable from the output directory.