Badge Application: Engine Programming (Bronze)

Jakub Rollo

1. Source Code Access

Repository URL: https://github.com/[redacted]
Gameplay video URL: https://www.youtube.com/watch?v=fGq35nRC-SE

Code Mapping (Feature Implementation):

- **Frame Rate & Timing Loops:** `GameCore.cpp` – Specifically the `Game::run()` function, which implements the main engine loop. It utilizes `sf::Clock` to manage fixed time-step logic updates (every 10ms) independent of the variable rendering frame rate.
- **Render Loop & Graphics Pipeline:** `Dungeon.cpp` – Inherits from `sf::Drawable`. The `Dungeon::draw()` function manages the rendering pipeline, issuing draw calls for the tilemap, enemies, and items using `sf::VertexArray` (primitive type `sf::Quads`) and custom shaders (`sf::Shader`).
- **Input Management System:** `GameCore.cpp` – The `getInput()` function acts as the central input manager, polling `sf::Event` structures and dispatching commands via `handleActions()` and `handleMouseClick()`.
- **Save & Load System:** `GameCore.cpp` – Implements serialization logic to write player stats, inventory data, and current level state to local save files, enabling persistent gameplay across sessions.
- **Pathfinding Algorithms (A*):** `Astar.cpp` – Contains the custom `AStar::findPath()` implementation. It utilizes a custom `Node` struct with f, g, h cost heuristics and operator overloading for priority queue management.
- **Procedural Generation (BST):** `DungeonGenerator.cpp` – Implements Binary Space Partitioning via the `BinaryRoom` class. It includes algorithms for `Split()`, `Trim()`, and connection logic via `AddCorridors()`.
- **Collision & Physics Interactions:** `Dungeon.cpp` – Handles grid-based physics interactions through `isWalkable()` for collision detection and `isPlayerWithinRadius()` for proximity triggers (interactions/combat).

2. Technical Summary

Implemented Features & Interaction

The engine is built in C++ using the SFML library to handle the low-level graphics and windowing layer. The core architecture relies on a custom `GameCore` class that acts as the central engine manager. This class implements the main game loop (`Game::run()`), which is responsible for coordinating the fixed time-step updates for game logic and variable time-step updates for rendering.

The `GameCore` processes raw input from the system (mapped to specific gameplay actions) and passes state changes to the `Dungeon` class. The `Dungeon` class serves as the container for the entity ecosystem, managing the render loop for the tilemap, dynamic enemies, and interactive items. This structure effectively separates the "Engine" control logic from the specific "Game" content simulation. Additionally, the build system utilizes CMake to handle cross-platform dependency management (vcpkg) and dynamic linking, ensuring the engine runs correctly on both Windows and Linux environments.

Algorithms Applied

Two major algorithms were implemented from scratch to handle spatial and navigational tasks:

- **Binary Space Partitioning (BST):** Adapted for procedural dungeon generation in `DungeonGenerator.cpp`. This algorithm recursively divides the available space to create varied, non-overlapping room layouts, ensuring a unique map for every playthrough. It makes extensive use of the `BinaryRoom` class to manage the hierarchical split data.
- **A* (A-Star) Pathfinding:** Implemented in `Astar.cpp`. This algorithm serves two distinct engine purposes:
    1. **AI Navigation:** Controlling enemy movement to intelligently navigate around obstacles toward the player.
    2. **Generation Analysis:** Calculating the "furthest point" metric during level generation (via `findFarthestOnes()`) to determine the optimal Start and Exit positions for the player.

Debugging & Profiling Strategy

Development utilized Visual Studio's native debugging and diagnostic tools to monitor stability and performance. A significant performance bottleneck was identified during the level generation phase.

The initial implementation of the "furthest point" calculation attempted to brute-force find the longest path between all possible combinations of floor tiles. Using performance timing logs and CPU profiling, I identified this as the slowest operation (approaching $O(N^2)$ complexity on large maps).

I addressed this by optimizing the search algorithm to restrict the search space, checking only points located in opposing halves of the dungeon (using 'isSurroundedByOnes' checks to filter invalid start nodes). This trade-off significantly reduced generation load times while maintaining a functional and enjoyable level layout.

3. Profiling Evidence

The following data was captured using Visual Studio Diagnostic Tools to analyze the application's runtime performance.

Performance Analysis: The profiling results indicate that the most significant performance bottlenecks are located within the frame rendering pipeline and file loading operations. As evidenced by the Critical Path analysis (Figure 2), the majority of execution time is consumed by `nvoglv64.dll` (NVIDIA OpenGL Driver) and system-level I/O calls. Since these overheads originate from external driver implementations and library-dependent resource loading—rather than the custom engine logic—they represent fixed costs that cannot be significantly optimized within the current scope of the project.

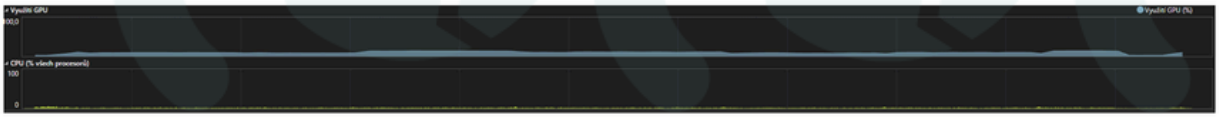

Figure 1: **Process Memory & CPU Timeline:** An overview of system resource consumption. The top graph indicates GPU utilization, while the bottom graph tracks CPU usage across all processors, verifying that the fixed time-step logic does not saturate the CPU.

Figure 2: **Critical Path Analysis:** A breakdown of CPU time by function. The analysis highlights that the heaviest load comes from the external graphics driver (`nvoglv64.dll`), confirming that the custom engine loop (`AlexandriasDungeonMaster`) is not the primary bottleneck.

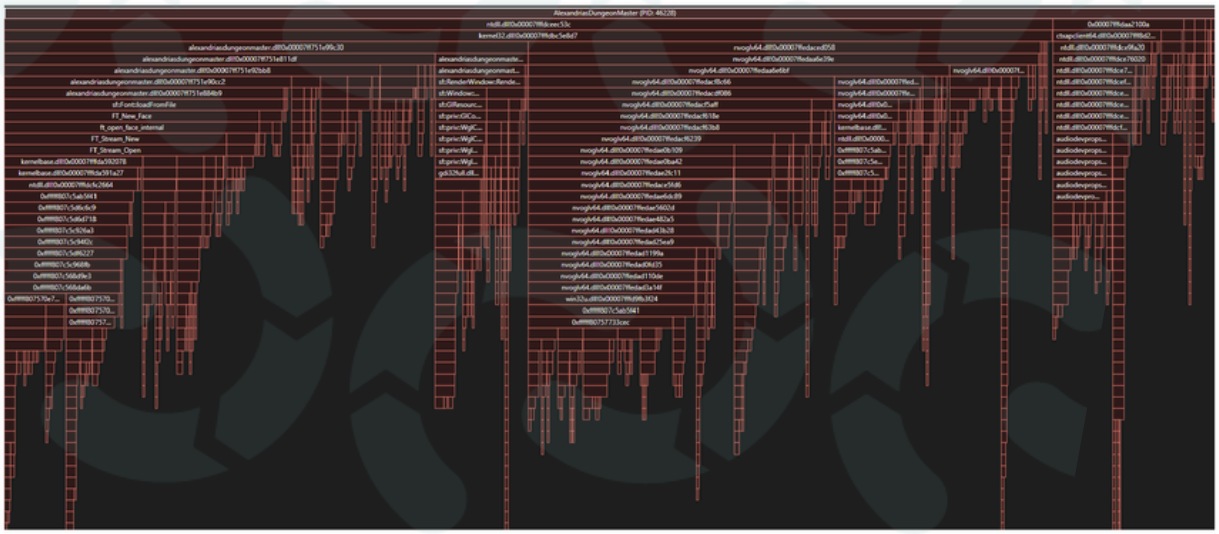

Figure 3: **Flame Graph Visualization:** A visual representation of the call stack depth and frequency. This view was used to trace the execution flow of the rendering pipeline and ensure that no recursive function calls were causing stack overflow or unnecessary overhead.

4. Selected Source Code Samples

4.1 Game Loop & Timing (GameCore.cpp)

```cpp
int Game::run(sf::RenderWindow& window) {
    window.setFramerateLimit(60);
    // ... Initialization code ...
    sf::Clock clock;
    sf::Time elapsed = clock.restart();

    while (window.isOpen()) {
        getInput(window, dungeon, player);
        if (isEndOfLevel) break;

        // Fixed time-step update loop
        if (gameState == GameState::Playing) {
            elapsed = clock.getElapsedTime();
            if (elapsed.asMilliseconds() >= 10) {
                dungeon.update();
                elapsed = clock.restart();
            }
        }

        // Variable time-step rendering
        window.clear(sf::Color::Black);
        window.draw(dungeon, &(dungeon.getShader()));
        player.DisplayInventory(window);
        window.display();
    }
    return 0;
}
```

4.2 Render Pipeline (Dungeon.cpp)

```cpp
void Dungeon::draw(sf::RenderTarget& target, sf::RenderStates states) const {
    auto [offsetX, offsetY] = calculateOffsets(states);

    // Adjust the view transform for camera movement
    states.transform *= sf::Transform().translate(-offsetX, -offsetY);

    // Apply the tileset texture
    states.texture = &m_tileset;

    // Draw the main vertex array (Dungeon Tiles)
    target.draw(m_vertices, states);

    // Draw entities
    drawItems(target, states);
    drawEnemies(target, states);
    drawPlayer(target, states);
    drawHPBars(target, states, offsetX, offsetY);
}
```

4.3 A* Pathfinding Algorithm (Astar.cpp)

```cpp
std::vector<std::pair<int, int>> AStar::findPath(int start_x, int start_y, int
    goal_x, int goal_y) {
    std::priority_queue<Node> open_set;
    // ... Vector initialization ...
```

```
    open_set.emplace(start_x, start_y, 0, heuristic(start_x, start_y, goal_x,
    goal_y));

    while (!open_set.empty()) {
        Node current = open_set.top();
        open_set.pop();

        if (current.x == goal_x && current.y == goal_y) {
            // Reconstruct path logic ...
            return path;
        }

        // Neighbor processing
        std::vector<Node> neighbors = getNeighbors(current, goal_x, goal_y);
        for (const auto& neighbor : neighbors) {
            if (!visited[neighbor.x][neighbor.y]) {
                open_set.push(neighbor);
                came_from[neighbor.x][neighbor.y] = { current.x, current.y };
            }
        }
    }
    return {};
}
```

5. Gameplay Screenshots

Figure 4: Gameplay screenshot 1: Main Menu

Figure 5: Gameplay screenshot 2: Inside the game, main player with no items and a skeleton enemy.

Figure 6: Gameplay screenshot 3: Inside the game after playing for a while.