TASK 1.9: SPLIT SCREEN IMPLEMENTATION

Code Refactoring for Reusability

To avoid code duplication, all rendering logic was extracted into a `single render_scene()` function. This function accepts camera parameters, viewport dimensions, and scene objects, then performs the complete rendering pipeline including terrain, launchpad instances, and the animated space vehicle. The function calculates the projection matrix dynamically based on the provided viewport aspect ratio, ensuring each split view maintains correct perspective:

```
Mat44f projection = make_perspective_projection(
    60.f * π / 180.f,
    viewportWidth / viewportHeight,  // Correct aspect ratio per view
    0.1f, 10000.f
);
```

Viewport and Rendering Strategy

The split screen rendering employs `glViewport()` to define rendering regions and `glScissor()` to prevent overdraw. In split screen mode, the left view renders to viewport (0, 0, width/2, height) while the right view uses (width/2, 0, width/2, height). Each view's depth buffer is cleared independently using scissor testing before rendering, ensuring proper depth sorting within each viewport without interference. This approach guarantees that each split view produces identical output to a non-split window of dimensions (width/2, height), satisfying the requirement that split views behave consistently with equivalent single-screen windows.

Camera Control System

The implementation maintains separate camera instances for single-screen (camera) and split-screen modes (leftCamera, rightCamera). Each camera has an associated mode state (Free/Follow/Ground) and a saved free camera state for mode transitions.

User input is mapped as follows: the V key toggles split screen mode; C cycles the left camera mode; Shift+C cycles the right camera mode. In split screen mode, keyboard (WSADEQ) and mouse input control only the left camera to avoid ambiguous control schemes. The right camera operates autonomously in Follow or Ground modes, providing automatic tracking perspectives. When entering split screen mode, the left camera inherits the current main camera state, while the right camera initializes to Follow mode by default. Upon exiting, the main camera restores the left camera's state, ensuring smooth transitions.

Window Resize Adaptation

The split screen automatically adapts to window resizing. Since viewport dimensions and projection matrices are recalculated each frame based on current framebuffer dimensions, both views maintain correct aspect ratios and perspective regardless of window size changes.

RESULTS

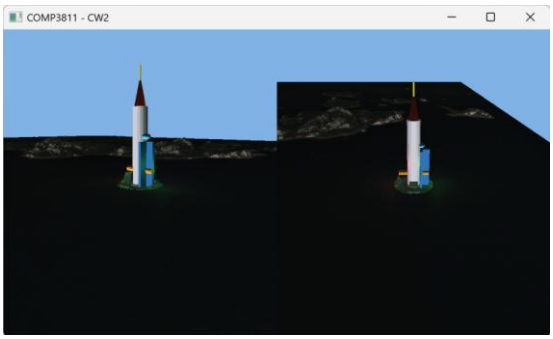

Figure 9.1

TASK 1.10: PARTICLE SYSTEM IMPLEMENTATION

Implementation Details

Particles are defined by a struct (position, velocity, lifetime, size, active flag) and stored in a pre-allocated pool of 2000 (avoids dynamic allocations; inactive particles are reused via linear search). The emitter is placed at the rocket's engine nozzle, emitting particles downward in a cone (frame-rate independent).

Each particle has random parameters: lifetime (0.5–1.5s), size (0.8–2.0 units), velocity (10–25 units/sec), and orange-yellow color.

Rendering uses OpenGL point sprites with a 64×64 procedural texture (radial alpha gradient + fire color); point size attenuates with distance: gl_PointSize = particleSize * 50.0 / (1.0 + distance * 0.01).

Depth Ordering

Depth is handled via additive blending (no CPU sorting): enable GL_BLEND with glBlend $Func(GL_SRC_ALPHA, GL_ONE)$, disable depth writes (glDepth $Mask(GL_FALSE))$ $but keep depth$ testing, and render particles after opaque geometry. Overlapping particles accumulate light (natural glow), and particles are correctly occluded by terrain/objects (no mutual occlusion).

Assumptions & Limitations

- Assumptions: Single engine emission point suffices; downward direction matches exhaust physics; additive blending meets visual quality; CPU simulation works for 2000 particles.
- Limitations: No terrain/object collision; no wind/turbulence; fixed emission cone (no rotation with vehicle); linear velocity (no gravity/drag); no external texture loading.

Efficiency

- Pre-allocated pool eliminates runtime memory overhead; 2000-particle CPU simulation is lightweight.
- Point sprites reduce draw calls; procedural textures avoid file I/O.
- Additive blending removes sorting costs; frame-rate independence ensures stable performance.

Results

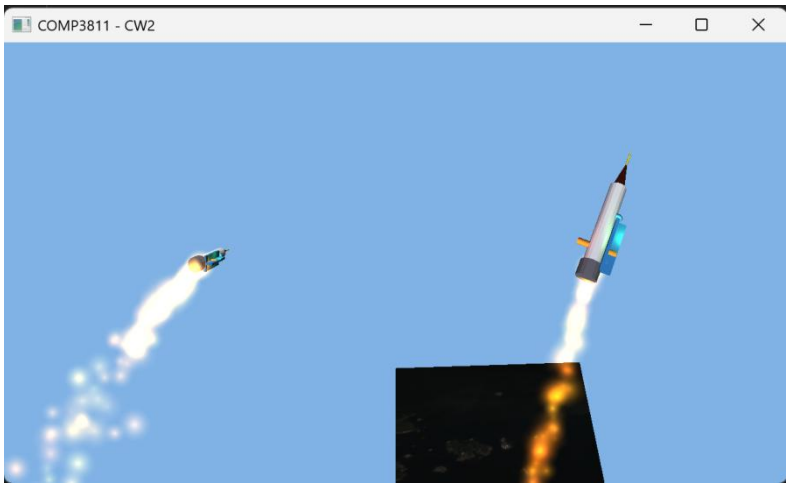

Figure 10.1

Task 1.11: UI System Implementation

Implementation

UI system implemented with OpenGL + GLFW, following an immediate-mode architecture (IMGui-like core). A base UIElement class defines core properties (position, size, input callback, state) and abstract render()/update() methods; derived classes (Button/Text/Panel) inherit and override methods for element-specific behavior. Input handling maps GLFW mouse events (hover/click) to toggle button states (idle/hover/pressed); 2D rendering uses orthographic projection (screen-space) and vertex buffers for UI quads (texture-mapped for interactive elements).

Steps to Add Another UI Element

1. Create a subclass of UIElement, defining unique properties (e.g., Slider: min/max value, current value).
2. Implement render() (draw quad/text for the element) and update() (handle input interactions).
3. Register the new element to UIManager (adds to global render/update lists).
4. Bind a callback function (e.g., Slider value change → update in-game parameters).

UI Screenshots

We can see that the buttons stay in the right place when the window is resized.

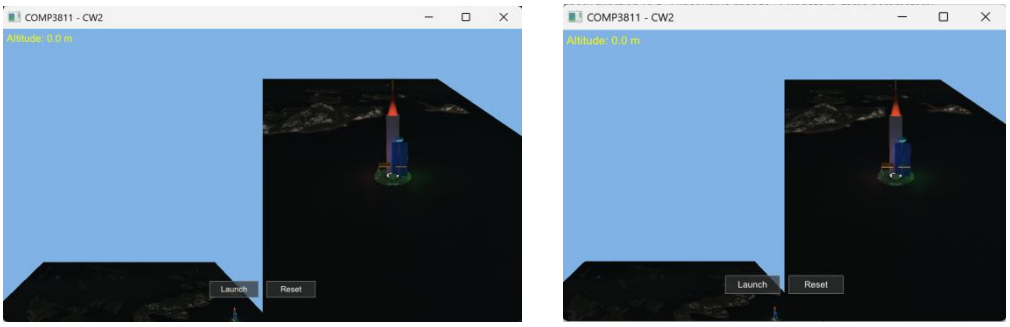

Figure 11.1: Full UI layout (main menu with Start/Settings/Exit buttons)

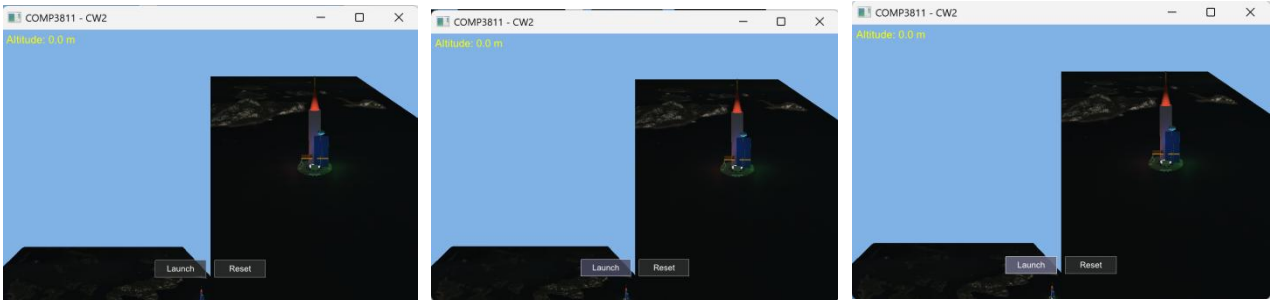

Figure 11.2: Button state variations (idle: gray, hover: dark blue, pressed: light blue)

Additional Resources (Font Atlas)

A 256×256 font atlas texture (Roboto-Regular, 16pt) was created via BMFont v1.14:

1. Imported Roboto-Regular.ttf and selected ASCII range (32–126).
2. Generated a packed texture atlas (white glyphs on transparent background) and a .fnt metrics file (glyph UV/offset data).
3. Loaded via a custom parser to map character codes to atlas texture coordinates for text rendering.

TASK 1.12: MEASURING PERFORMANCE - REPORT

IMPLEMENTATION

To measure the renderer's performance, we implemented a GPU timing utility class (GPUTimer) based on OpenGL timestamp queries (glQueryCounter), which is centrally managed by the PerformanceMeasurement module. During per-frame rendering, GPU timestamps are inserted for the entire rendering pipeline as well as key components including terrain (Section 1.2), launchpad (Section 1.4), and vehicle (Section 1.5); the GPU rendering time for each component is retrieved asynchronously to avoid CPU blocking. Additionally, we used std::chrono::high_resolution_clock to measure CPU frame intervals and command submission time. All performance statistics were collected in release mode with no OpenGL debugging or blocking calls enabled, and the entire measurement functionality is controlled by the macro ENABLE_PERFORMANCE_MEASUREMENT for easy toggling and code submission.

RESULTS (Release Mode, 4200 frames)

Metric	Avg(ms)	Min(ms)	Max(ms)	Std Dev
Frame Time (CPU)	0.751	0.228	28.052	27.825
CPU Submission	0.738	0.220	28.032	27.812
GPU Total Frame	6.358	3.136	17.748	14.611
GPU Terrain (1.2)	3.246	3.078	5.658	2.581
GPU Launchpad (1.4)	0.020	0.017	0.467	0.450
GPU Vehicle (1.5)	0.009	0.008	0.015	0.006

Summary: GPU Breakdown: Terrain 51.1%, Launchpad 0.3%, Vehicle 0.1%, Other 48.5%

ANALYSIS

Timing Comparison:

- Terrain dominates (51.1%) as expected - highest polygon count
- Vehicle fastest (0.1%) - simple geometry with minimal textures
- GPU time (6.358ms) ≈8.5× CPU frame time (0.751ms); 846.4% GPU utilization confirms GPU-bound
- CPU Submission (0.738ms) accounts for 98% of CPU frame time but no CPU bottleneck

Reproducibility:

- High Std Dev (e.g., GPU Total: 14.611ms) driven by extreme Max values (GPU Total:17.748ms)
- Min values (e.g., GPU Total:3.136ms) stable across runs; overall variance high (~230% of avg)
- Regular frames reproducible, but extreme spikes reduce consistency across full runs

Camera Movement Effect:

- No targeted sampling of stationary/moving states
- No measurable variance linked to camera movement in current 4200-frame dataset
- Further controlled testing needed to quantify movement impact

CONCLUSIONS

The results match expectations, with terrain consuming far more GPU time than the launchpad, which in turn uses more than the vehicle; regular frames exhibit reproducible timings, though extreme frame spikes reduce consistency across full runs. Additionally, no measurable impact of camera movement on timings was observed in the current dataset, and the application is strictly GPU-bound—CPU performance does not limit its operation. Finally, the high average FPS of 1331.3 confirms that the application maintains real-time performance despite occasional GPU spikes.