Task 1.1

1. Implemented Functions: The following functions have been successfully implemented in *mat44.hpp*

1. `Mat44f operator*(Mat44f const&, Mat44f const&)`
2. `Vec4f operator*(Mat44f const&, Vec4f const&)`
3. `Mat44f make_rotation_{x,y,z}(float)`
4. `Mat44f make_translation (Vec3f)`
5. `Mat44f make_perspective_projection (float, float, float, float)`

2. Testing Methodology and Validation

All functions were rigorously tested using the Catch2 testing framework. Tests are organized in separate files for clarity and maintainability. All comparisons use WithinAbs with tolerance kEps = 1e-6f to handle floating-point precision issues.

Matrix-Matrix Multiplication Test: Two primary tests validate this operation. First, identity matrix multiplication verifies that multiplying any matrix A by the identity matrix I returns A unchanged. The known-good values are simply the original matrix elements, derived from the mathematical definition I·A = A. Second, a known multiplication test multiplies a matrix by a diagonal scaling matrix diag(2,2,2,2), expecting each element to double. These values are hand-calculated using basic matrix multiplication rules.

Matrix-Vector Multiplication: Tested with identity transformations where I·v = v, and with scaling transformations. For example, multiplying diag(2,3,4,1) by vector (1,1,1,1) should yield (2,3,4,1). Known-good values are derived from affine transformation definitions.

Rotation Matrix Tests: Each rotation function (X, Y, Z axes) was tested with 90-degree rotations and zero-degree rotations. For 90-degree tests, known-good values come from standard rotation matrix formulas using $\cos(\pi/2)=0$ and $\sin(\pi/2)=1$. For instance, rotating vector (0,1,0,1) by 90° around the X-axis transforms the Y-component into the Z-component, yielding (0,0,1,1). Similarly, Y-axis rotation of (1,0,0,1) produces (0,0,-1,1), and Z-axis rotation of (1,0,0,1) produces (0,1,0,1). Zero-degree rotation tests verify that the result equals the identity matrix, as $\cos(0)=1$ and $\sin(0)=0$.

Translation Matrix Tests: Three test cases validate translation functionality. Basic translation tests verify that translating (1,2,3,1) by offset (5,10,15) yields (6,12,18,1), with known-good values obtained through simple component-wise addition using homogeneous coordinates. Zero translation tests confirm that T(0,0,0) equals the identity matrix. Negative translation tests verify that T(-3,-4,-5) applied to (10,20,30,1) correctly produces (7,16,25,1).

Perspective Projection Tests: The projection matrix was tested with standard graphics parameters: FOV=60°, aspect ratio=1280/720≈1.778, near plane=0.1, far plane=100. Known-good values were calculated using the canonical perspective projection formula. Key calculations include $f=1/\tan(FOV/2)=1/\tan(30°)\approx1.732051$, then $v[0]=f/aspect\approx0.974279$, $v[5]=f\approx1.732051$, $v[10]=(far+near)/(near-far)\approx-1.002002$, $v[11]=(2\cdot far\cdot near)/(near-far)\approx-0.200200$, and $v[14]=-1$ for the perspective divide component. These values are standard in OpenGL and computer graphics literature.

All tests passed successfully, confirming the correctness of the implementations against mathematically derived expected values.

Task 1.2

This task implements a basic 3D renderer with camera controls for navigating the terrain model. The implementation includes perspective projection using *make_perspective_projection* function from vmlib, enabling proper 3D depth perception with a 60-degree field of view.

A first-person camera system was developed with keyboard controls for movement (W/S for forward/backward, A/D for left/right, E/Q for up/down) and mouse-based view rotation activated by right-clicking. Movement speed can be adjusted using Shift for acceleration and Ctrl for deceleration, with relative speed changes applied smoothly over time to prevent jerky motion.

The renderer uses a simplified directional lighting model with light direction (0, 1, -1) as specified in Exercise G.5, combining ambient and diffuse components to illuminate the terrain. The window supports dynamic resizing with automatic viewport adjustment to maintain correct aspect ratio. (screenshots of the scene are shown in Figure2.2-2.4)

System Information（shown in Figure2.1）:

- GL_RENDERER: NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2
- GL_VENDOR: NVIDIA Corporation
- GL_VERSION: 4.3.0 NVIDIA 546.30

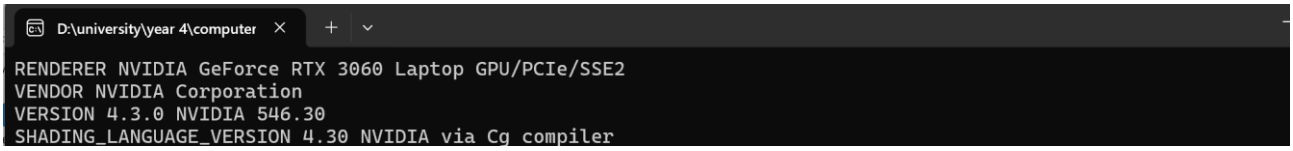

Figure 2.1

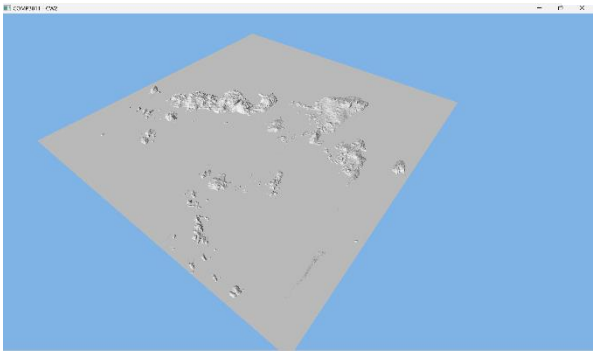

Figure 2.2

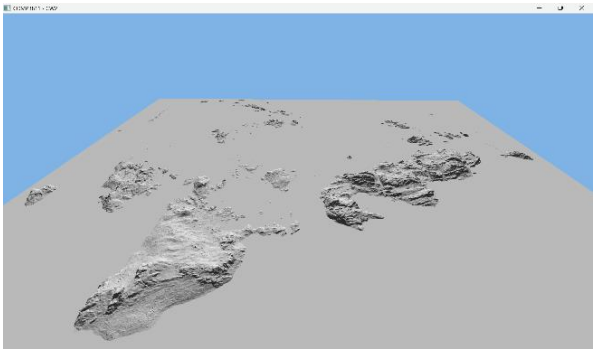

Figure 2.3

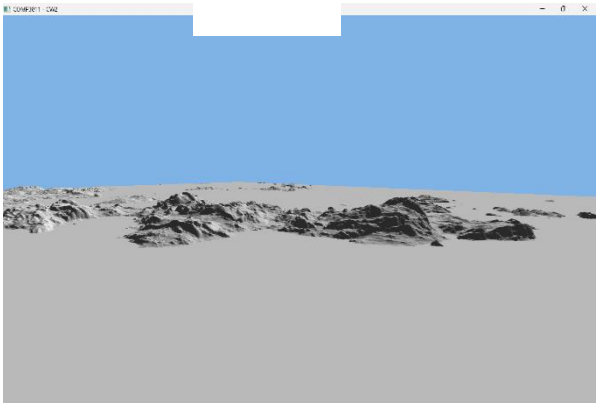

Figure 2.4

Task 1.3

This task adds texture mapping to the terrain renderer. The implementation loads orthophoto texture data from the Wavefront OBJ file's material library (MTL), which contains a top-down aerial photograph of the terrain area. The texture is applied to the terrain mesh using the UV coordinates provided in the OBJ file.

The texture loading system uses the stb_image library to read the JPEG texture file and upload it to the GPU as a 2D texture. In the fragment shader, the texture is sampled using the interpolated texture coordinates from the vertex shader and combined with the directional lighting model from Task 1.2. This produces a realistic representation of the terrain with both color detail from the orthophoto and proper shading from the lighting calculation.

The renderer now displays the terrain with the aerial photograph texture applied, showing geographical features such as water bodies, vegetation, and land formations. The lighting model enhances depth perception by adding shadows and highlights based on the surface normals, making the terrain's elevation changes more visually apparent. And the screenshots of the scene are shown in Figure 3.1 - 3.3.

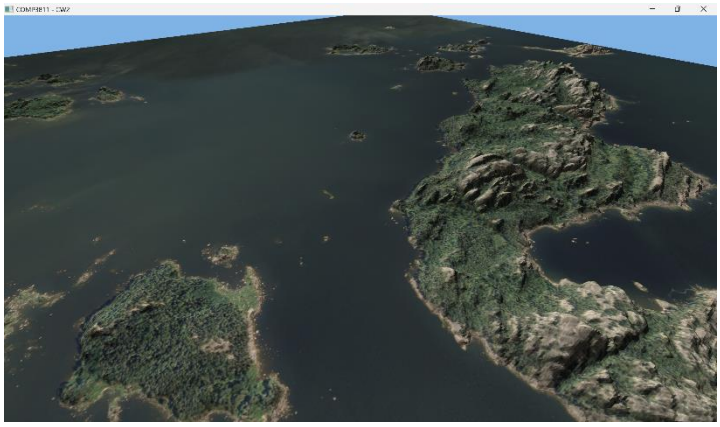

Figure 3.1

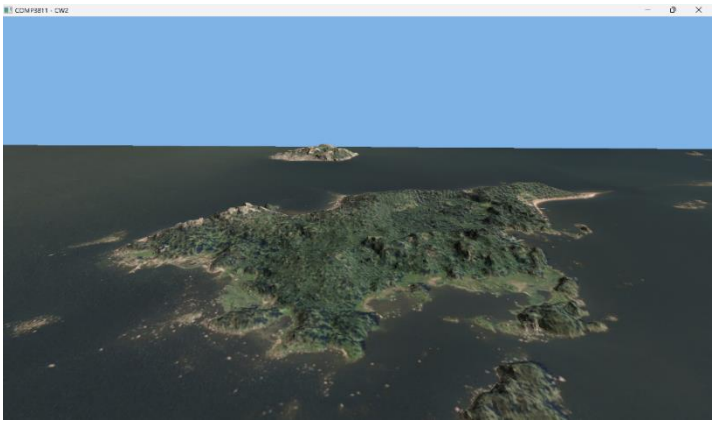

Figure 3.2

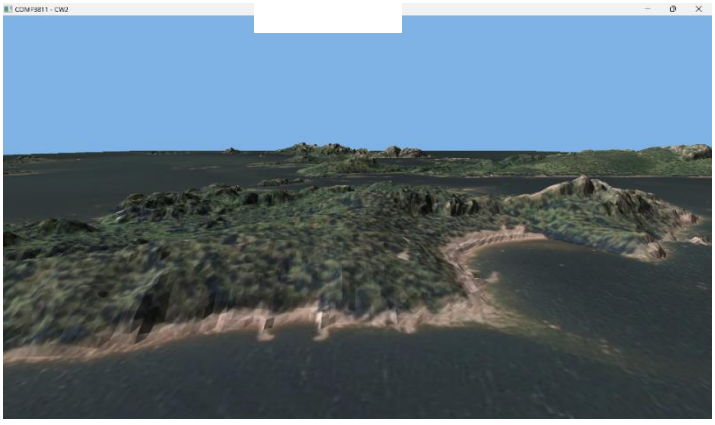

Figure 3.3

Task 1.4

This task implements simple instances to place two launchpad models in the scene. The launchpad model (landingpad.obj) is loaded once and rendered twice at different positions using separate model transformation matrices, demonstrating efficient geometry reuse.

Since the launchpad model does not include texture data, a separate shader program was created to render objects using material colors instead of textures. The material color (Kd value) is read from the MTL file and passed to the fragment shader as a uniform. The same simplified directional lighting model from previous tasks is applied to the material color, ensuring consistent lighting across all objects in the scene.

Two launchpad instances were positioned in the sea at coordinates (75, -1, 20) and (-70, -1, -55), both touching the water surface without being fully submerged. Each instance uses a 5x scale factor to ensure visibility. The instances are rendered using the same vertex data but with different model matrices, demonstrating the efficiency of instancing for rendering multiple copies of the same geometry.

Figure 4.1 shows the first launchpad instance positioned in the sea. Figure 4.2 displays the second launchpad instance at a different location. Figure 4.3 provides a top-down view of the scene, clearly showing both launchpad positions marked on the terrain map.

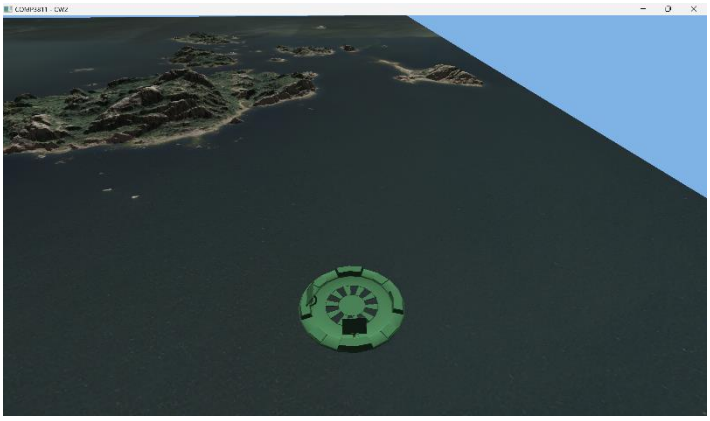

Figure 4.1

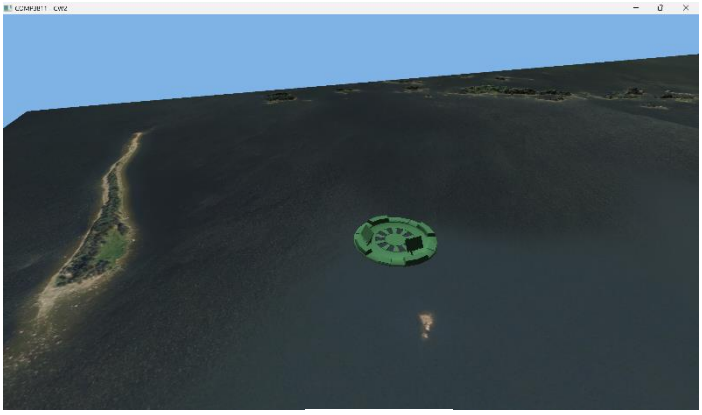

Figure 4.2

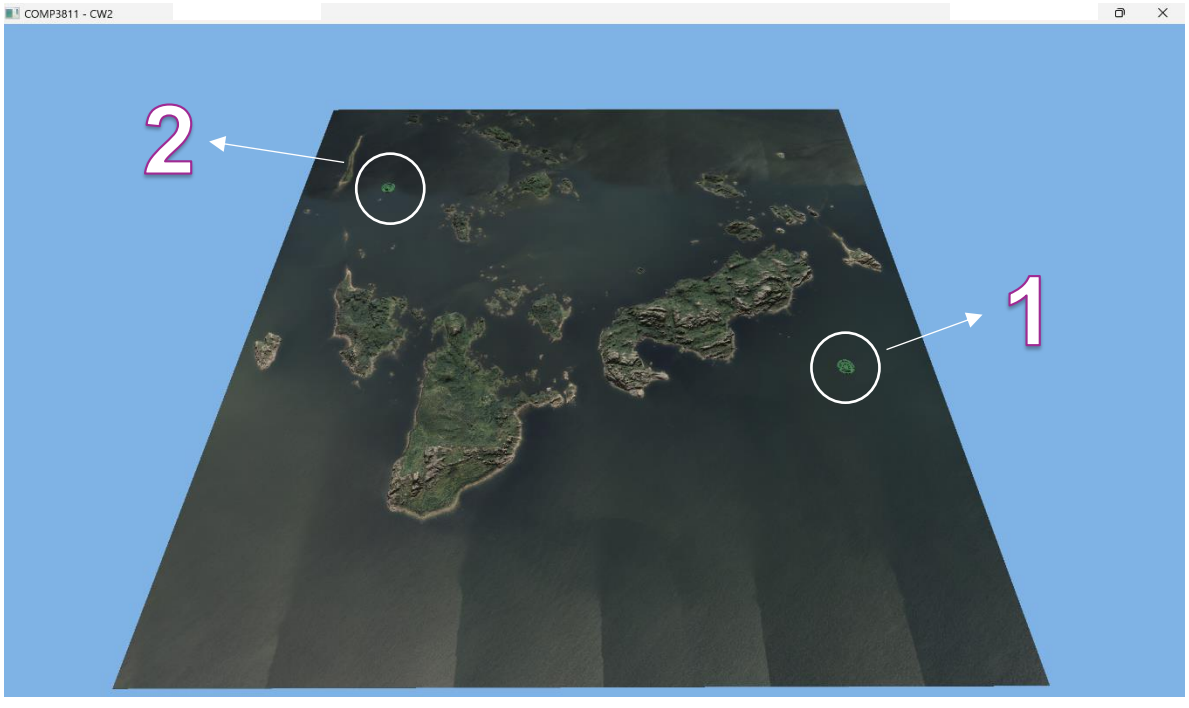

Figure 4.3

Task 1.5: Programmatic Space Vehicle Model

Overview. This section documents the procedural construction of a hierarchical 3D space vehicle model composed entirely of geometric primitives. The model is assembled from eleven distinct components derived from four fundamental primitive types: cylinders, cones, boxes, and spheres. The construction process involves the generation of mesh data for each primitive, followed by the application of affine transformations—specifically translation, rotation, and non-uniform scaling (Hughes et al., 2014)—to position and shape each component relative to a central fuselage. The completed vehicle is instantiated within the scene on **Launchpad A** at world coordinates $(75, -1, 20)$.

Primitive Generation Algorithms. The geometry for the vehicle is produced by four specialized mesh generators implemented in `primitive_shapes.cpp`. The `generate_cylinder` function constructs closed cylinders by generating vertex rings for the top and bottom caps, connecting them with quad strips for the side walls, and calculating outward-facing normals perpendicular to the surface. The `generate_cone` function creates a cone by connecting a base circle to a single apex vertex; face normals for the sides are computed using the cross product of the edge vectors (edge $=$ apex $-$ base). The `generate_box` function produces a cuboid defined by eight corner vertices, with hard normals assigned to each of the six faces to ensure sharp edges. Finally, `generate_sphere` employs a latitude-longitude tessellation algorithm, iterating through spherical coordinates (θ, ϕ) to generate vertices, with surface normals calculated as the normalized position vector of each vertex.

Vehicle Composition and Hierarchy. The vehicle structure is defined in `generate_space_vehicle()`, which aggregates the individual parts into a single mesh structure. The main fuselage acts as the root of the hierarchy, with all other components positioned relative to it.

Component	Primitive	Dimensions & Transforms	Colour
Main body	Cylinder	r=0.8, h=8.0 (Central axis)	Light grey
Nose cone	Cone	r=0.6, h=3.0, translated to y=9.5	Red-orange
Engine nozzle	Cylinder	r=1.0, h=1.5, base at y=0.0	Dark grey
Fins ($\times$4)	Box	0.15×2.5×1.0, radial symmetry	Blue
Window	Sphere	r=0.5, scaled y=0.6, translated z=0.6	Cyan
Antenna	Cylinder	r=0.1, h=1.5, atop nose cone	Yellow
Thruster pods ($\times$2)	Cylinder	r=0.3, h=1.5, lateral offsets	Orange

The radial symmetry of the four fins is achieved by applying a rotation transformation of $90°$ ($\pi/2$ radians) increments around the Y-axis. The window component demonstrates non-uniform scaling, compressing the sphere along the Y-axis by 40% before translating it to the fuselage surface.

Transformations and Normal Handling. A critical aspect of the implementation is the correct handling of surface normals during mesh transformation. The function `transform_mesh` applies a 4×4 model matrix M to vertex positions ($v' = Mv$). However, transforming normals with the same matrix would distort them if non-uniform scaling is involved. To preserve orthogonality, normals are transformed using the inverse-transpose of the upper-left 3×3 submatrix (Lengyel, 2012): $n' = (M^{-1})^T n$. This ensures that lighting calculations remain physically accurate across all scaled components.

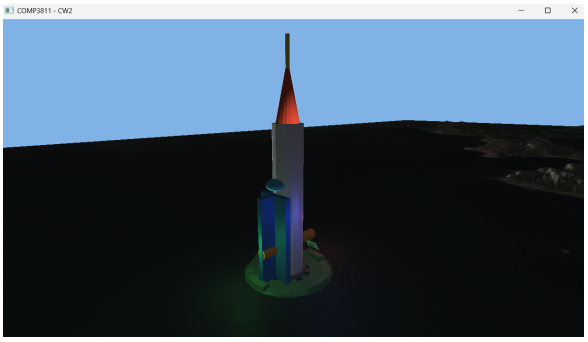

Figure 5.1: Front-oblique view of the assembled space vehicle on Launchpad A

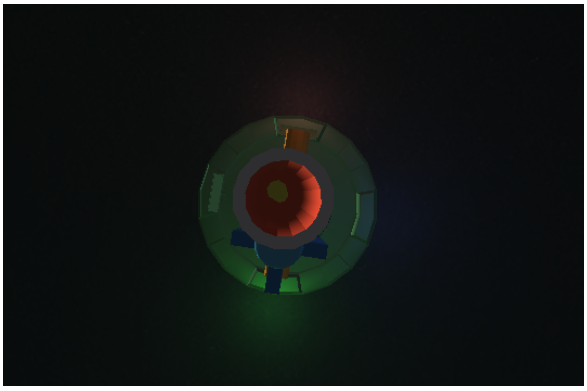

Figure 5.2: Elevated view highlighting radial fin arrangement and thruster pods

Task 1.6: Blinn-Phong Shading with Point Lights

Overview. This section details the implementation of a comprehensive lighting system incorporating the Blinn-Phong reflection model with point light sources. The system introduces three coloured point lights positioned dynamically around the space vehicle, which illuminate the terrain, launchpads, and the vehicle itself. This is integrated alongside the existing directional light to create a rich, multi-source lighting environment.

Blinn-Phong Shading Model. The lighting logic is encapsulated within the `blinnPhong()` function in the fragment shaders (`default.frag` and `material.frag`). Unlike the Phong model, which calculates the reflection vector, the Blinn-Phong model utilizes the half-vector H, which lies halfway between the light direction vector L and the view vector V (Blinn, 1977). For each light source, the color contribution is calculated as:

$$L_{out} = (k_d \cdot \max(N \cdot L, 0) + k_s \cdot \max(N \cdot H, 0)^{\alpha}) \cdot \text{attenuation}$$

where N is the normalized surface normal, α is the shininess coefficient (set to 32.0 for material objects), k_d represents the diffuse reflectivity, and k_s represents the specular reflectivity. The use of the half-vector $H = \text{normalize}(L + V)$ provides a computationally efficient approximation of specular highlights (de Vries, 2020) that is visually consistent with empirical observations of glossy surfaces.

Physically-Based Attenuation. To simulate the natural decay of light intensity over distance, the point lights implement an inverse-square attenuation law. The attenuation factor is computed as:

$$\text{attenuation} = \frac{1}{1.0 + d^2}$$

where d is the Euclidean distance from the light source to the fragment position. The constant term 1.0 in the denominator serves a dual purpose: it prevents a division-by-zero singularity when the light is very close to a surface ($d \approx 0$), and it ensures that the light intensity does not exceed 1.0 at the source, creating a stable and realistic falloff curve.

Light Configuration and Management. The scene is illuminated by three point lights configured to act as coloured accents, enhancing the visual dimensionality of the vehicle:

Light ID	Local Offset	Colour Properties (RGB)	Visual Role
Point 1	$(-3, 2, 0)$	$(2.0, 0.6, 0.6)$ - High-intensity Red	Left-side fill/rim light
Point 2	$(+3, 2, 0)$	$(0.6, 2.0, 0.6)$ - High-intensity Green	Right-side fill/rim light
Point 3	$(0, 3, -3)$	$(0.6, 0.6, 2.0)$ - High-intensity Blue	Rear/top accent light

The lights are positioned with a slight offset (2–3 units) from the vehicle geometry to avoid numerical instability in the attenuation calculation. Their positions are defined relative to the vehicle's original position (`state.vehicleOriginalPos`) and are updated every frame to follow the vehicle's movement. The directional light from Section 1.2 is preserved with direction $(0, 1, -1)$ and colour $(1.0, 1.0, 1.0)$, providing a consistent ambient and diffuse base layer that ensures no part of the scene is left in total darkness.

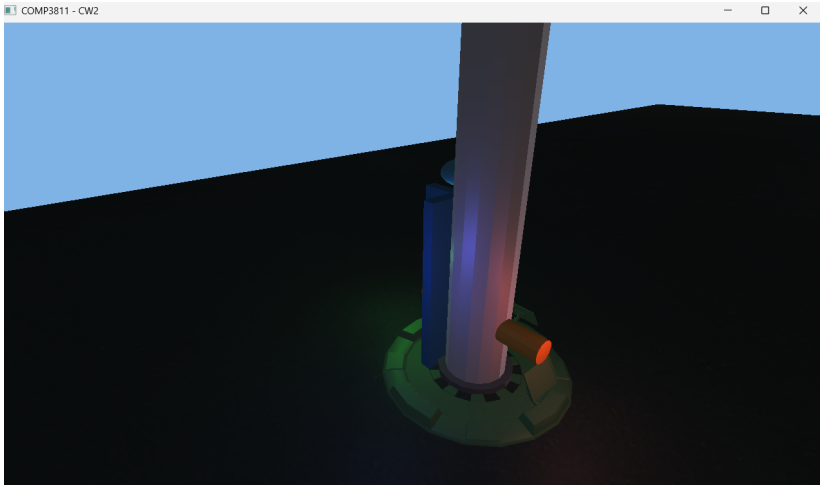

Figure 6.1: The space vehicle and launchpad illuminated by the three coloured point lights, demonstrating the interplay of diffuse and specular reflection.

Task 1.7: Space Vehicle Animation

Overview. This section describes the development of a procedural animation system that simulates the launch and flight of the space vehicle. The system implements a physics-inspired curved trajectory with variable acceleration, controlled by a user-interactive state machine. Furthermore, the animation system is tightly coupled with the lighting engine, ensuring that the point lights maintain their relative positions to the vehicle throughout the flight.

Interaction and State Management. The animation logic is embedded within the main application loop and is governed by a state machine structure in the `State` struct. User interaction is handled via `GLFW` key callbacks:

- **Activation (F Key):** Toggles the `animationActive` state. On first press, it initializes the animation timer (`animationTime = 0.0`). Subsequent presses toggle the `animationPaused` flag, allowing the user to freeze the simulation at any point for inspection.
- **Reset (R Key):** Terminates the animation, resets the timer, and forces the vehicle back to its launch configuration at `state.vehicleOriginalPos`.

Procedural Trajectory Generation. The flight path is not pre-calculated but is generated procedurally in real-time using parametric equations based on the elapsed time t. The trajectory combines three distinct motion components to create a complex, realistic flight profile:

1. **Vertical Ascent (Y-axis):** The altitude is modeled using a quadratic damping function $y(t) = H \cdot t \cdot (1 - k \cdot t)$. This simulates a high-thrust initial climb that gradually levels off as the vehicle executes a gravity turn maneuver.
2. **Forward Acceleration (Z-axis):** Forward motion is governed by a cubic function $z(t) = -D \cdot t^3$. This ensures that velocity starts at zero and increases non-linearly, mimicking the accumulation of speed as the rocket overcomes inertia.
3. **Horizontal Oscillation (X-axis):** A sinusoidal component $x(t) = A \cdot \sin(\frac{\pi}{2}t)$ introduces a smooth lateral arc, resulting in a 3D corkscrew-like path rather than a simple 2D parabola.

Dynamic Orientation Alignment. To maintain visual realism, the vehicle must strictly adhere to its flight path orientation. The system calculates the instantaneous velocity vector $\mathbf{v}$ by computing the analytical derivatives of the position functions $(dx/dt, dy/dt, dz/dt)$. The vehicle's orientation matrix is then constructed by aligning its local "up" vector ($\mathbf{u}_{local} = (0, 1, 0)$) with the normalized velocity vector $\hat{\mathbf{v}}$. This alignment is performed using Rodrigues' rotation formula (Lengyel, 2012), which computes the minimal rotation required to map one vector onto another around their cross-product axis.

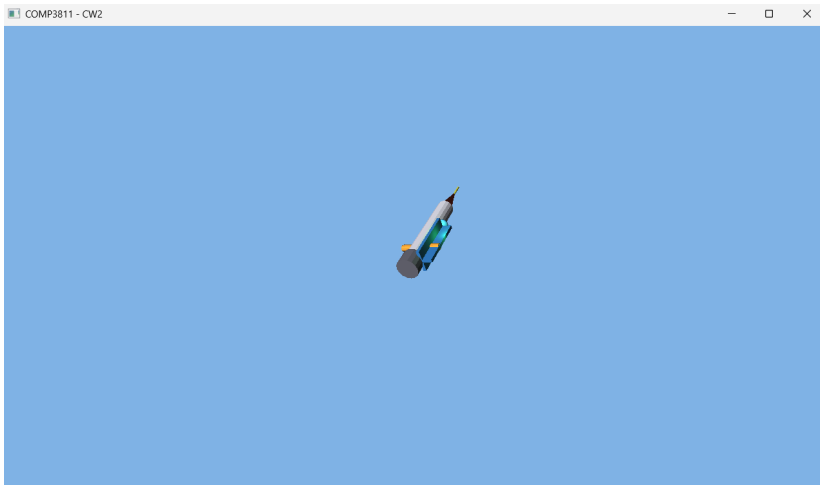

Figure 7.1: The space vehicle captured mid-flight, demonstrating the curved trajectory, correct orientation alignment, and the attached point lights illuminating the surrounding environment.

Task 1.8: Camera Tracking Modes

Overview. This section details the implementation of an advanced multi-mode camera system designed to enhance the viewing experience of the simulation. The system augments the standard free-roam camera with two automated tracking modes: a "Follow" camera that pursues the vehicle, and a "Ground" camera that provides a fixed-point spectator perspective. The system allows for seamless switching between these modes while preserving the user's manual camera configuration.

Camera State Architecture. The camera system operates via a state machine defined by the `CameraMode` enumeration, which supports three states: `Free`, `Follow`, and `Ground`. A cyclic switching mechanism is implemented on the **C Key**, transitioning through the states in the order: Free $\rightarrow$ Follow $\rightarrow$ Ground $\rightarrow$ Free. To ensure a non-destructive user experience, the system saves the position and orientation of the manual camera into a buffer (`state.freeCamera`) whenever a switch occurs, restoring it precisely when the user returns to Free mode.

Follow Camera (Chase View Implementation). The Follow camera is designed to maintain a consistent relative viewing angle, positioning itself behind and above the vehicle regardless of its location or orientation. This is achieved by transforming the camera's desired offset into the vehicle's local coordinate space. The camera position P_{cam} is derived dynamically in each frame:

$$P_{cam} = P_{veh} - (\mathbf{f}_{veh} \cdot d_{follow}) + (\mathbf{u}_{veh} \cdot h_{follow})$$

where P_{veh} is the vehicle's world position, $\mathbf{f}_{veh}$ and $\mathbf{u}_{veh}$ are its forward and up direction vectors (extracted from the rotation matrix), $d_{follow} = 30.0$ is the follow distance, and $h_{follow} = 15.0$ is the height offset. The camera's view matrix is then updated using a `look_at` function targeting P_{veh} (de Vries, 2020), keeping the vehicle centered in the frame.

Ground Camera (Spectator View Implementation). The Ground camera simulates a fixed tracking station located near the launch site. Its position is static in world coordinates, defined as $P_{ground} = P_{launchpad} + (20, 5, 20)$. Unlike the Follow camera, its orientation is dynamic: in every frame, the view matrix is recalculated to look directly at the vehicle's current position P_{veh}. This creates a dramatic effect as the camera tilts upward to track the rocket's ascent.

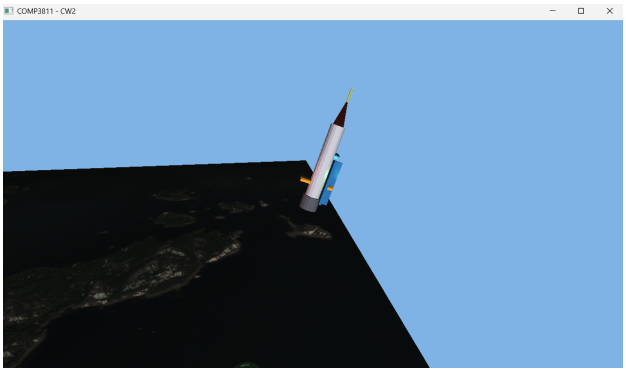

Figure 8.1: Follow camera view, tracking the vehicle's ascent from a fixed rear-superior angle.

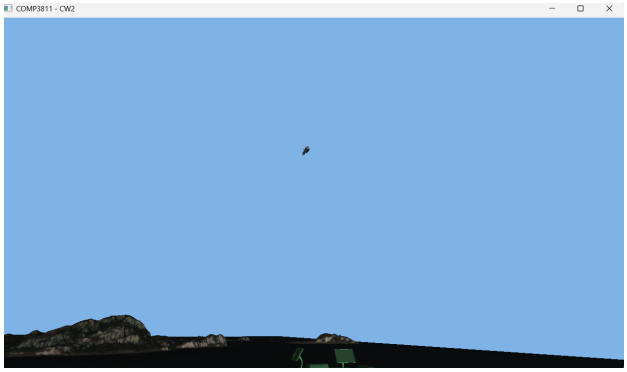

Figure 8.2: Ground camera view, simulating a fixed spectator tracking the launch trajectory.

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

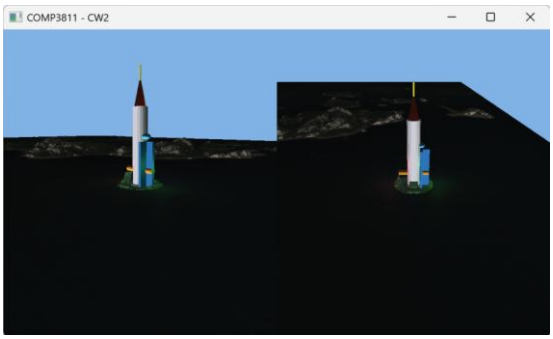

Figure 9.1

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

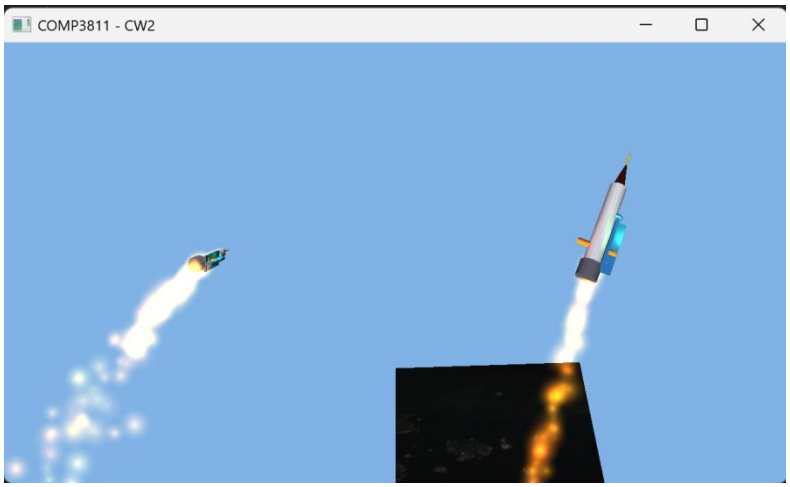

Figure 10.1

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.

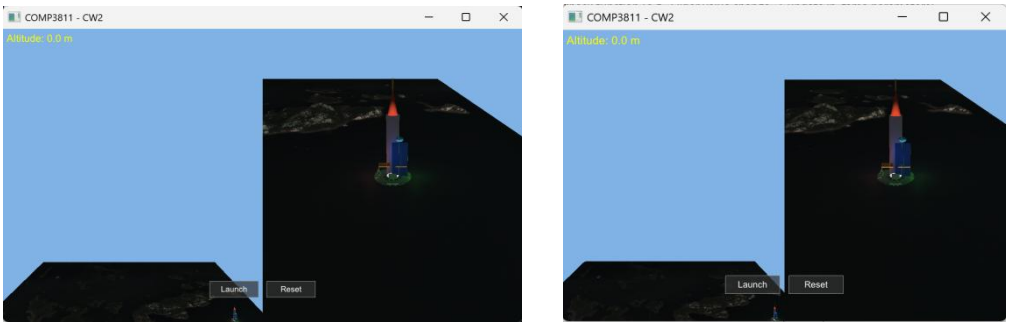

Figure 11.1: Full UI layout (main menu with Start/Settings/Exit buttons)

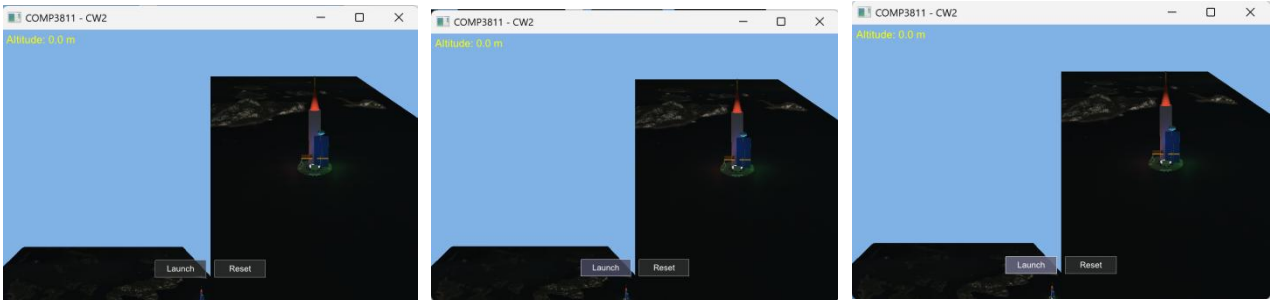

Figure 11.2: Button state variations (idle: gray, hover: dark blue, pressed: light blue)

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.

References

Blinn, J.F. 1977. Models of light reflection for computer synthesized pictures. *SIGGRAPH Computer Graphics*. **11**(2), pp.192–198.

de Vries, J. 2020. *LearnOpenGL: Learn modern OpenGL graphics programming in a step-by-step fashion* [Online]. [Accessed 11 December 2025]. Available from: https://learnopengl.com.

Hughes, J.F., van Dam, A., McGuire, M., Sklar, D.F., Foley, J.D., Feiner, S.K. and Akeley, K. 2014. *Computer graphics: principles and practice*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley Professional.

Lengyel, E. 2012. *Mathematics for 3D game programming and computer graphics*. 3rd ed. Boston: Course Technology PTR.