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	$0.15{\times}2.5{\times}1.0$, 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 $4{\times}4$ 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 $3{\times}3$ submatrix (Lengyel, 2012): $n' = (M^{-1})^T n$. This ensures that lighting calculations remain physically accurate across all scaled components.

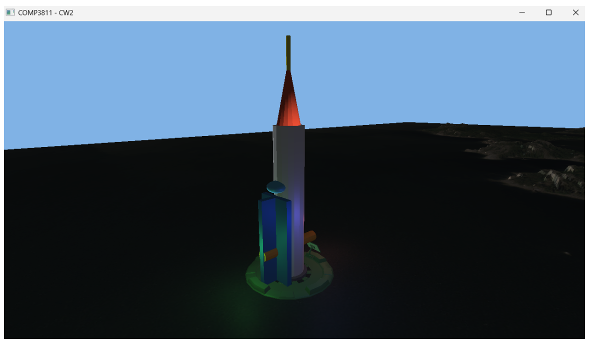

Figure 5.1: Front-oblique view of the assembled space vehicle on Launchpad A

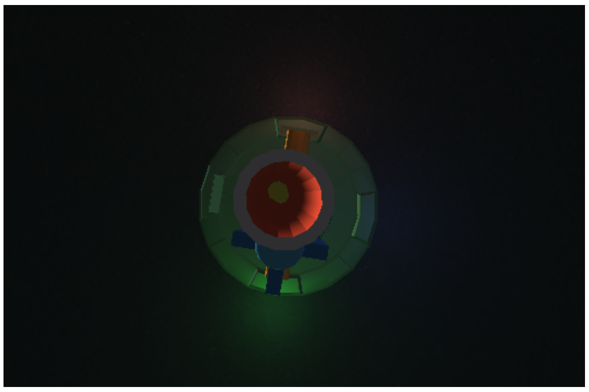

Figure 5.2: Elevated view highlighting radial fin arrangement and thruster pods

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.

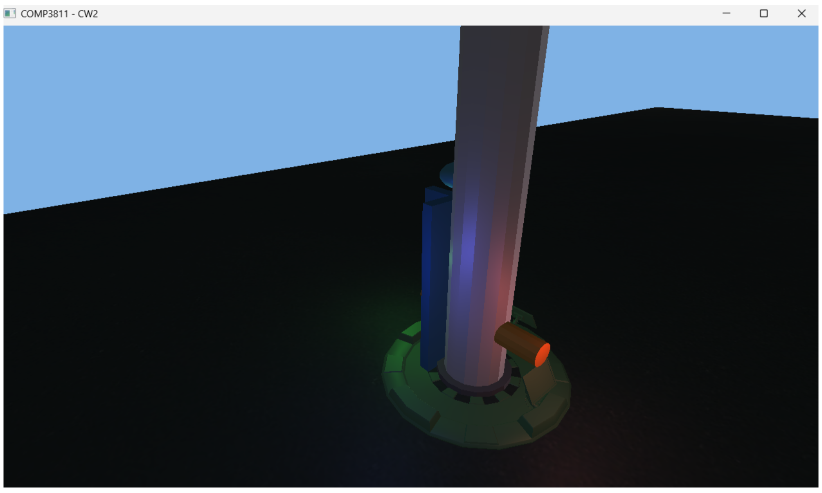

Figure 6.1: The space vehicle and launchpad illuminated by the three coloured point lights, demonstrating the interplay of diffuse and specular reflection.

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.

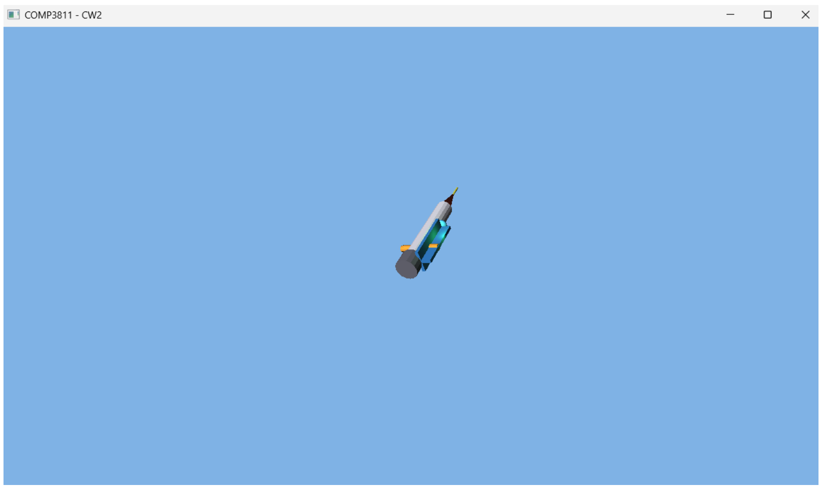

Figure 7.1: The space vehicle captured mid-flight, demonstrating the curved trajectory, correct orientation alignment, and the attached point lights illuminating the surrounding environment.

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.

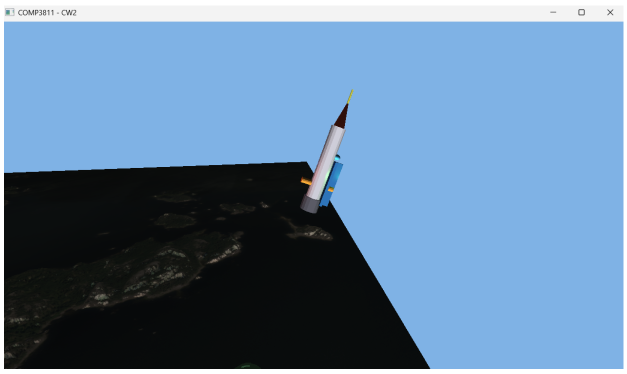

Figure 8.1: Follow camera view, tracking the vehicle's ascent from a fixed rear-superior angle.

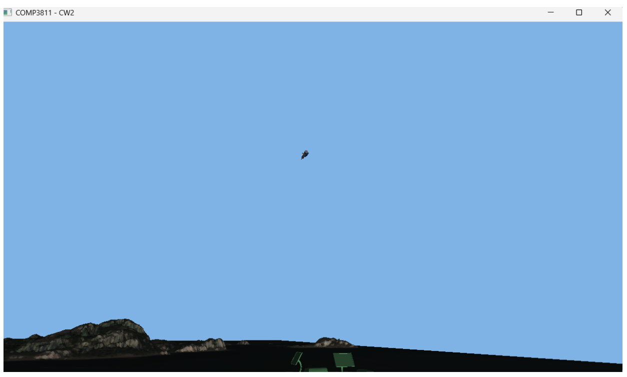

Figure 8.2: Ground camera view, simulating a fixed spectator tracking the launch trajectory.

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.