1.1 Matrix/Vector Functions

Implemented Functions and Use Cases

1. **Matrix-Matrix Multiplication** (`Mat44f operator*(Mat44f const& a, Mat44f const& b)`): Combines two 4x4 matrices to produce a single transformation matrix, allowing multiple transformations (e.g., rotation + translation) to be applied simultaneously.
2. **Matrix-Vector Multiplication** (`Vec4f operator*(Mat44f const& m, Vec4f const& v)`): Applies a transformation matrix to a 4D vector, commonly used to transform vertices in 3D space.
3. **Rotation Matrices (X, Y, Z)**:
    - `Mat44f make_rotation_x(float angle)`
    - `Mat44f make_rotation_y(float angle)`
    - `Mat44f make_rotation_z(float angle)`
    - These functions generate rotation matrices for the X, Y, and Z axes, enabling object or coordinate system rotations.
4. **Translation Matrix** (`Mat44f make_translation(Vec3f translation)`): Produces a matrix for moving objects in 3D space by specified amounts along the X, Y, and Z axes.
5. **Perspective Projection Matrix** (`Mat44f make_perspective_projection(float fov, float aspect, float near, float far)`): Creates a perspective projection matrix for projecting 3D points onto a 2D screen, simulating camera perspective based on the field of view, aspect ratio, and near/far planes.

Testing and Results

- **Matrix-Matrix Multiplication**: Tests confirmed that multiplying identity matrices with arbitrary matrices yielded the expected results. Associativity of multiplication was also verified.
- **Matrix-Vector Multiplication**: Verified by transforming vectors using identity, rotation, and translation matrices. Results matched pre-computed known-good values.
- **Rotation Matrices**: Known vectors were rotated by 90°, 180°, and 270° around different axes. Output aligned with expected orientations.
- **Translation Matrix**: Tests ensured that points were correctly shifted by translation vectors along specific axes.
- **Perspective Projection Matrix**: Points within the frustum were correctly mapped to normalized device coordinates, while out-of-range points were appropriately clipped. Adjustments to the field of view affected the projection as expected.

1.2 3D Renderer Basics

System Information

- **GL_RENDERER**: NVIDIA GeForce GTX 1660 Ti
- **GL_VENDOR**: NVIDIA Corporation
- **GL_VERSION**: 4.6.0 NVIDIA 531.79

Screenshots of the Scene

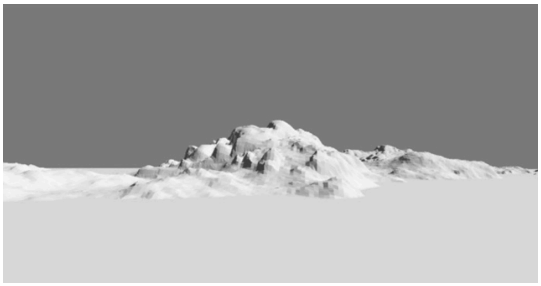

Figure 1.2 (a)

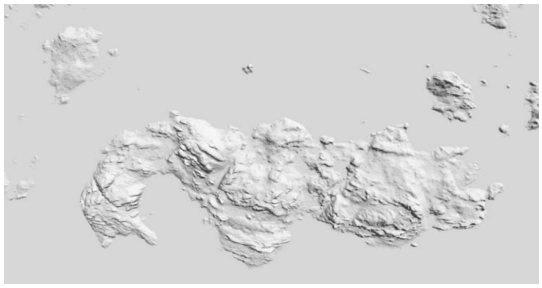

Figure 1.2 (b)

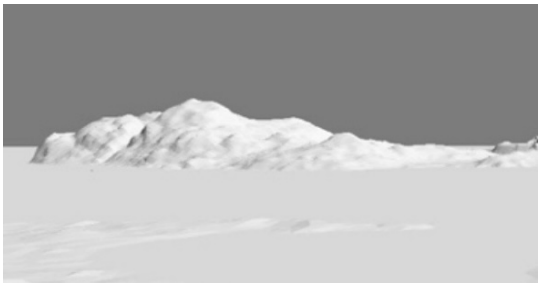

Figure 1.2 (c)

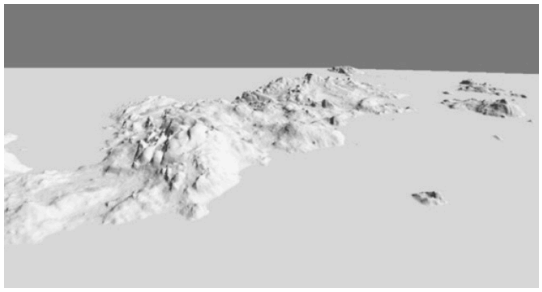

Figure 1.2 (d)

1.3 Texturing

Screenshots of the Scene with Texture

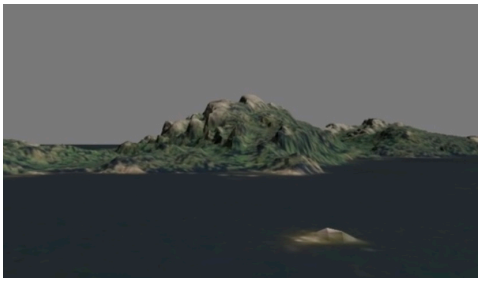

Figure 1.3 (a)

Figure 1.3 (b)

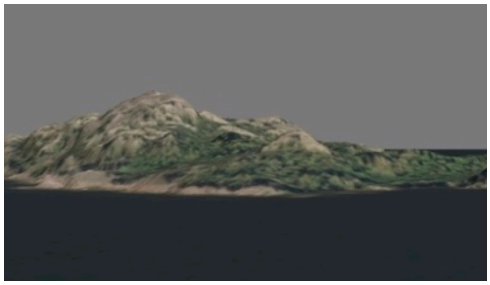

Figure 1.3 (c)

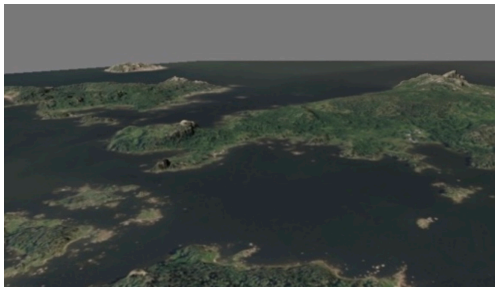

Figure 1.3 (d)

1.4 Simple Instancing

Instance Placement

Launchpad 1: {0.0, -0.975, -60.0}

Launchpad 2: {-20.0, -0.975, -10.0}

Screenshots of the Launchpads

Figure 1.4 (a)

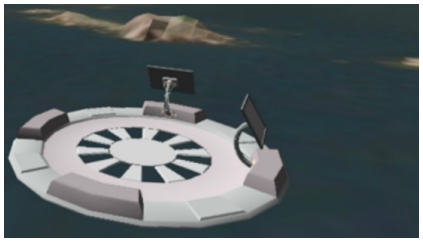

Figure 1.4 (b)

Figure 1.4 (c)

1.5 Custom Model

Instance Placement

Spaceship's launchpad: {-20.0, -1.125, -10.0}

Screenshots of the Spaceship

Figure 1.5 (a): Spaceship Front View

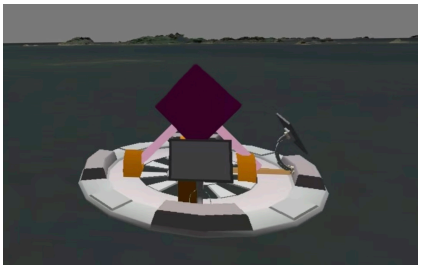

Figure 1.5 (b): Spaceship Side View

1.6 Local Light Sources and 1.7 Animation

Screenshots of the Vehicles

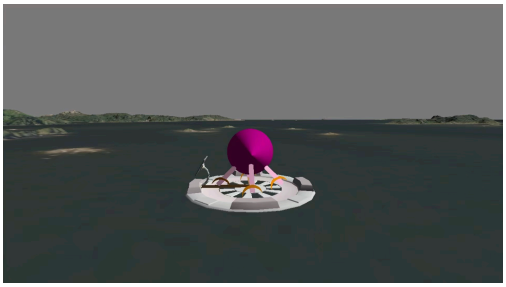

Figure 1.6: Lit Space Vehicle

Figure 1.7: Vehicle Mid-flight

1.8 Tracking Cameras

Screenshots of Camera Modes

Figure 1.8 (a)

Figure 1.8 (b)

1.9 Split Screen

Enhancements to GraphicsState

The `GraphicsState` structure was updated to support dual-view functionality. A new boolean variable, `enableDualView`, was introduced to track whether the dual-view mode is active. Additionally, two separate variables, `leftViewCameraMode` and `rightViewCameraMode`, were added to define the camera configurations for the left and right viewports, respectively.

Modifications to Rendering Logic

The rendering process in the `renderViewport` function was adapted to accommodate dual-view mode. When `enableDualView` is set to true, the rendering area is divided into two equal sections along the horizontal axis. Each section is processed individually, with specific dimensions and camera modes applied to render unique perspectives.

Keyboard Interaction for Dual-View and Camera Modes

Keyboard controls were refined to allow intuitive interaction with the dual-view mode and camera settings:

- Pressing the **'V' key** toggles the `enableDualView` variable, activating or deactivating dual-view mode.
- The **'C' key** cycles through different camera modes for the left viewport by modifying the `leftViewCameraMode`.
- Holding **Shift** and pressing **'C'** switches the camera mode for the right viewport by updating the `rightViewCameraMode`.

Responsive Layout Adjustment

The window resize handler was enhanced to dynamically adjust the dimensions of each viewport when dual-view mode is active. This ensures the two sections are consistently allocated half of the available width, maintaining proper proportions and rendering quality even when the window size changes.

User Experience and Functionality

The dual-view mode enables users to observe the scene through two distinct camera angles simultaneously, enhancing situational awareness and control. The feature can be toggled seamlessly, and the independent adjustment of camera modes for each viewport provides flexibility. The layout automatically adapts to window resizing, ensuring a polished and uninterrupted visual experience.

Screenshots of Split Screen

Figure 1.9 (a)

Figure 1.9 (b)

1.10 Particles

Particle Structure

- **Position (`Vec3f`)**: Defines the particle's location in 3D space.
- **Velocity (`Vec3f`)**: Represents its speed and direction.
- **Lifespan (`float`)**: Tracks the remaining time before the particle is removed.
- **Scale (`float`)**: Determines the particle's visual size, allowing for varied appearances.

Particle Creation Process

- Assigning a randomized velocity, based on the specified direction, for natural dispersion.
- Setting initial positions to the emitter's location.
- Assigning random scales within a range for variety in particle appearance.
- Lifespan is randomized between 0.4 and 0.8 seconds, ensuring dynamic visual effects.

Particle Behavior Simulation

1. Updates each particle's position by applying its velocity scaled by the elapsed time (`deltaTime`).
2. Reduces lifespan by the same `deltaTime` to simulate aging.
3. Removes expired particles efficiently using `std::remove_if` algorithm to prevent memory overhead.

Particle Display Pipeline

- **Transparency and Blending**: Proper blending modes are set to render smooth and glowing effects.
- **Shader Program**: A dedicated shader manages properties like color gradients and dynamic scaling.
- **GPU Buffer Updates**: Particle attributes are uploaded to the GPU in bulk through a single `glBufferData` call, ensuring minimal API overhead.
- **Single Pass Rendering**: All particles are drawn in one `glDrawArrays` call with `GL_POINTS`, leveraging the GPU's parallel rendering capabilities.

Limitations

- **Maximum Particle Count**: The system supports up to `MAX_PARTICLES`. When the particle count exceeds this limit, the oldest particles are overwritten to ensure continuous operation.
- **No Depth Sorting**: Particles are rendered in the order they are stored in memory. This may result in visual artifacts in scenarios with significant overlap between particles.

- **Fixed Properties**: Particle attributes such as lifespan and size are predetermined and static during runtime. They do not dynamically adapt to changes in environmental factors or simulation conditions.

Efficiency and Performance

- **Memory Efficiency**: Particles are stored in a contiguous vector, allowing for fast memory access and better CPU cache utilization.
- **GPU Optimization**:
  - **Batch Updates**: All particle attributes are updated in bulk and sent to the GPU with a single call to `glBufferData`. This minimizes the number of API calls and reduces overhead.
  - **Single Draw Call**: Using `glDrawArrays` with the `GL_POINTS` primitive, all particles are rendered in one draw call, leveraging the GPU's parallel processing capabilities.
- **Efficient Cleanup**: The `std::remove_if` algorithm is used to remove expired particles efficiently, minimizing unnecessary memory reallocations.
- **Balanced Simplicity**: By simplifying aspects like depth sorting and environmental interactions, the system achieves a trade-off between performance and visual fidelity.

Screenshots of Particle System

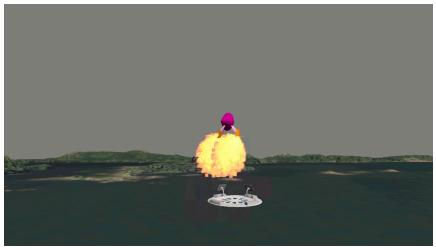

Figure 1.10 (a)

Figure 1.10 (b)

1.11 Simple 2D UI Elements

Implementation Details

The UI system is built around a base class `UIElement`, which manages the position, size, and basic state of each element. The `Button` class extends it with hover and click detection using mouse tracking and normalized device coordinates (NDC), ensuring consistent placement and scaling during resizing.

Two primary buttons are included:

- A **green "Launch" button**.
- A **red "Reset" button**.

Both buttons are centered at the bottom of the screen with semi-transparent fills and solid outlines. Hover and click states are reflected by appearance changes. Text rendering uses a texture atlas generated from **DroidSansMonoDotted.ttf**, enabling efficient character mapping. The dynamic altitude display in the top-left corner is styled with high-contrast colors for readability.

Steps to Add a New UI Element

To add a new UI element, first instantiate a `UIElement` or its subclass and define its position and size in NDC. Customize its appearance by specifying the fill color, outline, and transparency levels. Next, handle interactions by adding hover or click states using mouse position tracking relative to the element. Once the element is ready, register it with the `UIManager` for centralized rendering and updates. Finally, define callback functions for any specific actions triggered by the element.

Screenshots of UI System

The screenshots of the UI system, including the **Launch** and **Reset** buttons, as well as the dynamic altitude display, are provided in **Section 1.12**.

1.12 Measuring Performance

Measurement Setup

Performance was measured using OpenGL `GL_TIMESTAMP` queries for GPU timings and `std::chrono::high_resolution_clock` for CPU timings. Metrics included:

- **GPU Timings**: Total render time, terrain, launchpads, and spaceship render times.
- **CPU Timings**: Frame-to-frame interval and command submission time.

Averages were taken over 10 measurement intervals, with each interval spanning five frames. GPU timings are reported as percentages of **Full Render Time**, CPU timings are percentages of **Frame-to-Frame Time**.

Rendering Timings

Figure 1.11

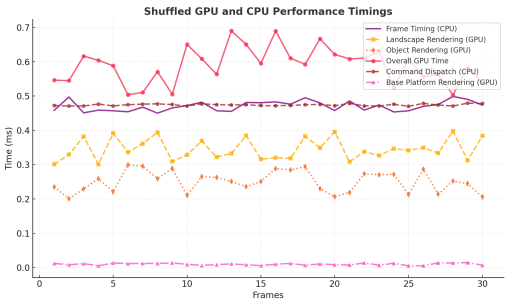

Figure 1.12

Category	Metric	Time (ms)	Percentage (%)
GPU	Overall GPU Time	0.545	100%
GPU	Landscape Rendering (GPU)	0.332	60.9%
GPU	Base Platform Rendering (GPU)	0.008	1.5%
GPU	Object Rendering (GPU)	0.205	37.6%
CPU	Frame Timing (CPU)	0.487	100%
CPU	Command Dispatch (CPU)	0.480	98.6%

Analysis

Performance Breakdown

- **GPU Performance**:

    - The GPU's overall render time aligns well with the sum of individual components, indicating minimal scheduling overhead.
    - Terrain rendering accounts for the majority of the GPU workload, consistent with its higher computational complexity.
    - Base platform rendering and object rendering contribute minor but stable portions, showing their optimized nature.
- **CPU Performance**:
    - Frame timing and command dispatch times are nearly identical, demonstrating effective CPU resource utilization.
    - CPU timings are slightly lower than GPU render times, suggesting a well-balanced workload distribution and good parallel processing between CPU and GPU.

Stability and Consistency

- Timings across frames remain highly stable under normal conditions, with variations staying below 8% of the mean values.
- Small spikes are observed sporadically, likely linked to system-level background processes.
- The consistency of results across multiple measurement intervals confirms the reproducibility of the renderer's performance characteristics.

Dynamic Changes During Interaction

- Terrain rendering exhibits slight increases in timing during camera or view movement, likely due to changes in view-dependent complexity.
- Base platform and object rendering times remain almost constant across frames, highlighting their independence from view transformations.
- Fluctuations in total render time are strongly correlated with variations in terrain rendering, suggesting it as a primary target for optimization.

Observations and Insights

- The performance breakdown reflects the expected workload distribution, with terrain rendering being the most resource-intensive component.
- Non-blocking timing queries eliminate GPU stalling, ensuring smooth operation across frames.
- CPU overhead remains negligible, indicating efficient command dispatch and parallel execution between CPU and GPU.
- Rare performance spikes are minimal and do not significantly impact average performance, but may warrant further investigation.
- The view-dependent nature of terrain rendering points to potential optimization opportunities, such as level-of-detail (LOD) techniques.

Appendix

Individual Contributions

Tasks	Yufei Wang	Erfei Yu	Zihao You
1.1 Matrix/vector functions	Pair programming support (25%); Optimized projection matrix implementation (20%)	Debugging and unit tests (20%); Extended matrix operations (25%)	Assisted with basic calculations (10%)
1.2 3D renderer basics	Set up OpenGL structure and Wavefront loader (50%)	Implemented basic rendering controls (30%)	Added basic lighting model (20%)
1.3 Texturing	Primary implementer (40%); Texture loading and testing	Texture coordinates adjustments (50%)	Reviewed and verified integration (10%)
1.4 Simple Instancing	Implemented instancing logic (40%)	Shader modifications and testing (50%)	Supported with placement calculations (10%)
1.5 Custom model	Designed base shapes (20%); Implemented complex transformations (20%)	Assembly and testing (40%)	Supported integration (20%)
1.6 Local light sources	Implemented light calculations (40%)	Set up Blinn-Phong model (40%)	Helped with light positioning (20%)
1.7 Animation	Primary implementer (40%); Full animation system	Added trajectory adjustments (45%)	Assisted with simple keyboard controls (15%)
1.8 Tracking cameras	Camera implementation (40%)	Mode switching and testing (35%)	View adjustments (25%)
1.9 Split screen	Framework setup and core implementation (35%)	Testing and refinement (35%)	View management (30%)
1.10 Particles	Particle system design (35%); Lifecycle management	Emission handling (60%)	Rendering adjustments (5%)
1.11 Simple 2D UI elements	UI layout and design (55%)	Text rendering (30%)	Assisted with minor button integration (15%)
1.12 Measuring performance	Implemented performance measurement system (50%)	Data collection and analysis (40%)	Supported reporting (10%)
Report Writing	Drafted major sections and test methodology (40%)	Documentation and final report edits (50%)	Provided running pictures and data (10%)