Coursework 2

Ayesha Rahman, ID:201522771
Sandra Guran, ID: 201538092
Geeyoon Lim, ID:201553023

Task 1.1 Matrix/vector functions

Introduction

This report details the implementation and testing of essential matrix and vector functions in the vmlib, specifically within the vmlib/mat44.hpp file. These functions are crucial for various computations in the coursework, demanding precise and reliable performance.

Testing Methodology

Each function was tested using the Catch2 library, focusing on comparing outputs against known-good values, derived either manually or via third-party libraries. This method effectively verifies the correctness of the implemented functions.

Matrix-Matrix Multiplication Test

Objective: Ensure matrix multiplication adheres to algebraic rules and handles real-world data.

Method: Two 4x4 matrices, matA and matB, were defined with predetermined values; their product was computed and compared against a manually computed expected result.

Verification: Each element of the result was compared against the expected values using Catch2's REQUIRE and Approx matchers.

Matrix-Vector Multiplication Test

Objective: Confirm compliance with mathematical standards in matrix-vector interactions**.**

Method: A fixed 4x4 matrix and a 4-dimensional vector were used, with the multiplication result compared against a pre-calculated expected outcome.

Verification: Elements of the resulting vector were validated against the expected values using Catch2's precision handling tools.

Rotation Matrix Tests

Objective: Verify accuracy of rotation matrices for x, y, and z axes at various angles.

Method: Rotation functions were tested with angles like 0, 90, and -90 degrees. The resulting matrices were compared to expected rotation matrices.

Verification: Elements of the results were checked against expected values using Catch2's WithinAbs matcher.

Translation Matrix Test

Objective: Confirm correct generation of translation matrices from vectors.

Method: A translation matrix was created from a specified vector and compared to an expected matrix.

Verification: The resulting matrix elements were compared to the expected matrix using Catch2's REQUIRE assertions.

Perspective Projection Matrix Test

Objective: Ensure 3D to 2D projections are accurate.

Method: Projection matrices were generated using standard parameters and compared with pre-calculated values.

Verification: The resulting projection matrix was validated against expected values using Catch2's WithinAbs matcher.

Conclusion

The implemented functions are essential for various graphical computations and transformations. The added tests, by comparing against known-good values, provide a robust validation method, ensuring the correctness and reliability of these fundamental functions in vmlib. This testing strategy lays a solid foundation for future work, ensuring that the core computational elements of the coursework are dependable and accurate.

Task 1.2 3D renderer basics

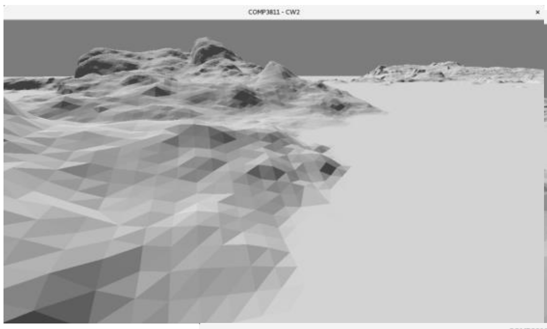
Introduction

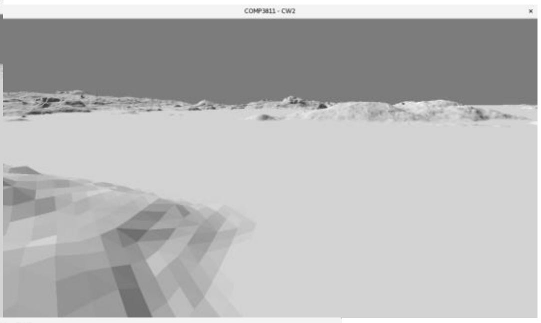
This report outlines the development and implementation of a 3D rendering application using modern shader-based OpenGL. The objective was to create a program capable of loading and displaying a Wavefront OBJ file representing a terrain near the Paarlahti region in Finland. The implementation includes a first-person style 3D camera controlled by keyboard and mouse inputs, along with a simplified directional lighting model for shading.

Methodology

1. OpenGL Initialization:
   GLFW library was used to create and manage the rendering window, while GLAD was employed to access OpenGL functionalities.
2. Shader Setup:
   Vertex and fragment shaders were loaded from external files ("default.vert" and "default.frag") to handle vertex transformations, lighting calculations, and fragment colouring.
3. Camera Control:
   Implemented a 3D camera allowing movement via WSAD keys, mouse control for view direction, and shift/ctrl keys for adjusting movement speed. GLFW callback functions were utilised for responsive and frame-rate independent camera control.
   Users can modify the camera's speed with the Shift and Control keys, which proves essential for different exploration needs within the environment. The Shift key increases the speed for faster movement, while the Control key slows it down for more precise navigation. These speed adjustments ensure that users can effectively adapt to various situational requirements.
   The system also includes functionalities for rotation and movement, controlled through keyboard inputs. The A/D keys are used for lateral rotation, and the E/Q keys for vertical movements, such as looking up and down. This setup provides an intuitive method for camera orientation, crucial for a seamless exploration experience. Additionally, the team implemented a boundary constraint by setting a maximum radius, preventing the camera from moving outside the predetermined environment limits.
4. Mesh Loading and Rendering:
   Loaded the Wavefront OBJ file representing the terrain near Paarlahti using a specialised function load_wavefront_obj. Created Vertex Buffer Objects (VBOs) and Vertex Array Objects (VAOs) to manage the mesh data for rendering. Utilised shaders to render the loaded mesh applying a simplified directional lighting model for shading purposes.
5. Matrices and Transformations:
   Calculated essential matrices such as the projection matrix and model-view matrix for rendering the scene. Projection matrix was generated using the make_perspective_projection function from vmlib. Model-view matrix was computed to handle camera movements and orientation.
6. Event Handling:
   Utilised GLFW callback functions to manage key presses for camera control and handle window events for smooth user interaction.
7. Debugging and Cleanup:
   Implemented error handling mechanisms, debug output, and proper cleanup procedures to ensure a stable and reliable application.

Results





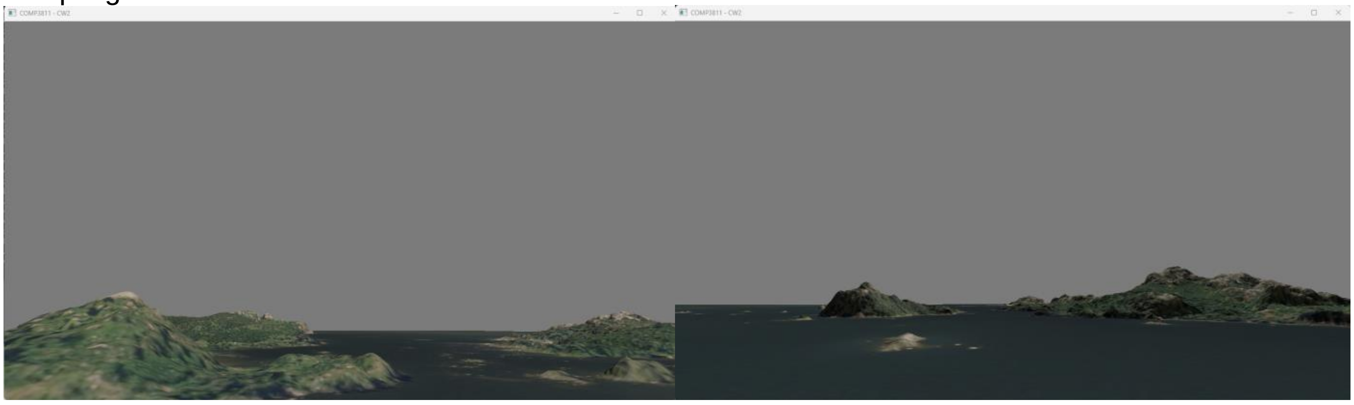
COMP3811 - CW2

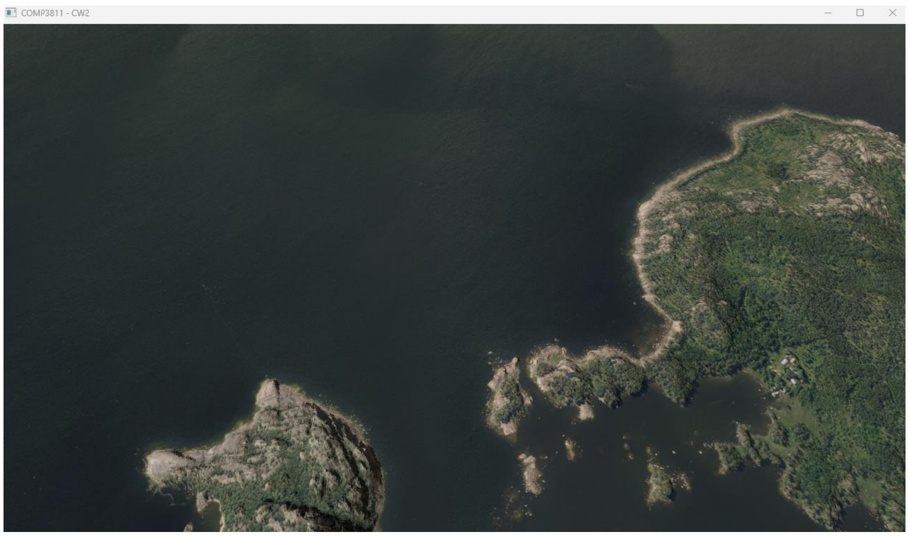
OpenGL Information:
System 1:

GL_RENDERER: llvmpipe (LLVM 16.0.6, 256 bits)
GL_VENDOR: Mesa
GL_VERSION: 4.5 (Core Profile) Mesa 23.1.4
SHADING_LANGUAGE_VERSION: 4.50
Renderer: llvmpipe is a software rasterizer that runs on the CPU. It's often used as a fallback when no GPU-accelerated renderer is available. This means that rendering tasks on this system are CPU-bound, which typically results in lower performance compared to hardware-accelerated (GPU) rendering.
OpenGL Version 4.5: This is a relatively recent version of OpenGL, suggesting good support for modern OpenGL features.
Shading Language 4.50: This version of GLSL is capable of advanced shading techniques, but the effectiveness will be limited by the CPU-bound nature of the renderer.
System 2
GL_RENDERER: Intel ® Iris ® Xe Graphics
GL_VENDOR: Intel
GL_VERSION: 4.3.0 - Build 31.0.101.4575
SHADING_LANGUAGE_VERSION: 4.30 - Build 31.0.101.4575
Renderer: Intel Iris Xe Graphics is an integrated GPU known for its relatively strong performance in the integrated GPU category. It should provide better rendering performance than llvmpipe due to hardware acceleration.
OpenGL Version 4.3.0: Slightly older than System 1 but still capable of handling most modern OpenGL features.
Shading Language 4.30: This version is capable of advanced shading but might lack some newer features present in 4.50.
System 3
GL_RENDERER: Intel(R) UHD Graphics
GL_VENDOR: Intel
GL_VERSION: 4.3.0 - Build 30.0.101.3111
SHADING_LANGUAGE_VERSION: 4.30 - Build 30.0.101.3111
Renderer: Intel UHD Graphics is another integrated GPU. It's generally less powerful than the Iris Xe Graphics, so you might expect slightly lower performance compared to System 2.
OpenGL Version 4.3.0 and Shading Language 4.30: Similar to System 2, it supports a good range of modern OpenGL features but might not have the latest advancements.
Comparative Analysis
Performance Expectations: System 1, using llvmpipe, is likely to have the lowest graphical performance due to its reliance on CPU for rendering. Systems 2 and 3, with hardware-accelerated rendering, should outperform System 1 in graphics-intensive tasks.
Feature Support: All systems support a version of OpenGL that is capable of modern graphics functionalities. However, the software rasterizer in System 1 might not utilise these capabilities as efficiently as the hardware-accelerated systems.
Use Case Suitability: Systems 2 and 3 are more suitable for applications requiring real-time graphics rendering, such as gaming or graphical simulations. System 1 could be used for development or testing purposes where high performance is not critical.
Conclusion
Based on the OpenGL specifications, System 1 is suitable for non-intensive graphics tasks or as a fallback, while Systems 2 and 3 are better suited for tasks requiring more graphical processing power. The choice between Systems 2 and 3 would depend on the specific performance needs and the relative importance of the slightly better capabilities of the Iris Xe Graphics over the UHD Graphics.
The developed 3D rendering application successfully achieved the objective of loading and displaying a Wavefront OBJ file representing the terrain near the Paarlahti region using modern shader-based OpenGL. The implementation includes a responsive first-person style 3D camera, allowing users to explore the rendered scene efficiently. The simplified directional lighting model added depth to the visualisation, highlighting the terrain features.
In summary, all three systems mentioned have varying hardware configurations but support modern OpenGL versions. They have different rendering capabilities due to their respective GPU models and vendors. System 1 relies on a software renderer, while Systems 2 and 3 use integrated Intel graphics, with slight variations in versions and builds. Despite differences, all systems successfully handled the 3D rendering application with varying OpenGL capabilities.

Task 1.3 Texturing

Objective
The objective of this phase of the project was to correctly map orthophotos onto a terrain mesh using texture mapping in OpenGL. This process involved ensuring the correct loading of texture images, associating texture coordinates with the mesh vertices, and modifying the rendering pipeline to incorporate texture sampling.





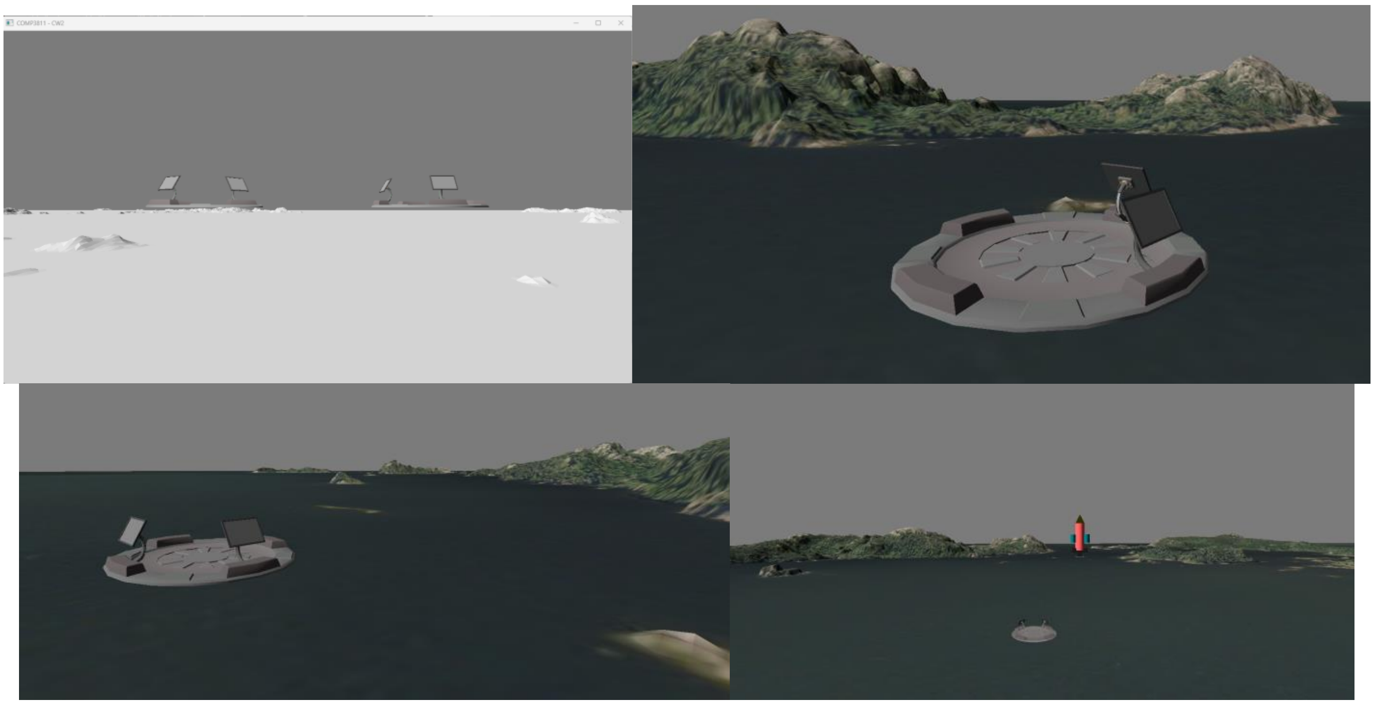
Methodology
The implementation followed a multi-step approach to integrate texture mapping into the existing OpenGL renderer:
Texture Loading: Utilised a custom function load_texture_2d that loads the JPEG image and generates a texture object in OpenGL. The texture parameters for wrapping and filtering were set to appropriate values to handle the texture display on the mesh correctly.
Shader Update: The vertex shader (default.vert) and fragment shader (default.frag) were updated. The vertex shader was adjusted to pass texture coordinates to the fragment shader. The fragment shader was modified to sample the texture using these coordinates and combine the sampled colour with the lighting calculations.
Buffer and Array Object Setup: The vertex buffer objects (VBOs) and vertex array object (VAO) were configured to include texture coordinates. The attribute pointers were set up to correctly pass the texture coordinate data to the shader.
Rendering Loop Changes: The rendering loop was updated to activate the texture unit, bind the loaded texture, and set the shader's sampler uniform to point to the correct texture unit before initiating the draw call.
Debugging and Testing: Throughout the implementation, various checks for OpenGL errors were made to catch any issues early on. Testing was conducted using simple textures to confirm the correctness of texture coordinates before using actual orthophotos.
Results
The final outcome was that the terrain mesh was successfully textured with the orthophoto. The texture coordinates from the Wavefront OBJ file were correctly mapped onto the mesh, and the orthophoto was displayed in accordance with the terrain's geometry.
Challenges Encountered
- The initial renderings displayed a white mesh, indicating issues with texture sampling or coordinate mapping.

- Debugging was required to ensure that the shaders were receiving and using the correct texture coordinates.
- Adjustments were made to the texture loading function to handle non-power-of-two textures and ensure compatibility with various image formats.

Solutions and Troubleshooting
- The following solutions were applied to address the challenges:
- Validation of texture loading by temporarily using a checkerboard texture to confirm correct texture application.
- Shader code was reviewed and corrected to ensure proper passing and sampling of texture coordinates.
- OpenGL texture parameters were set to use GL_LINEAR filtering and GL_CLAMP_TO_EDGE for wrapping to accommodate the full range of texture coordinates.

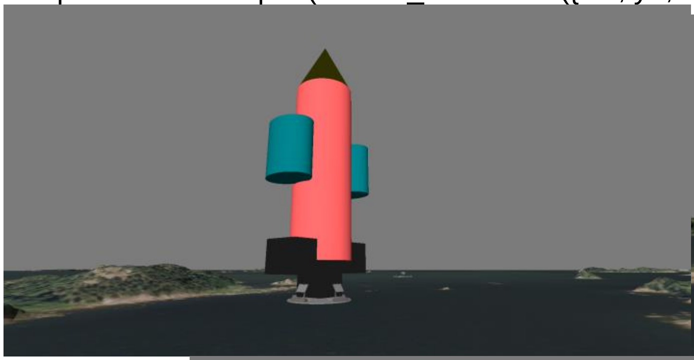
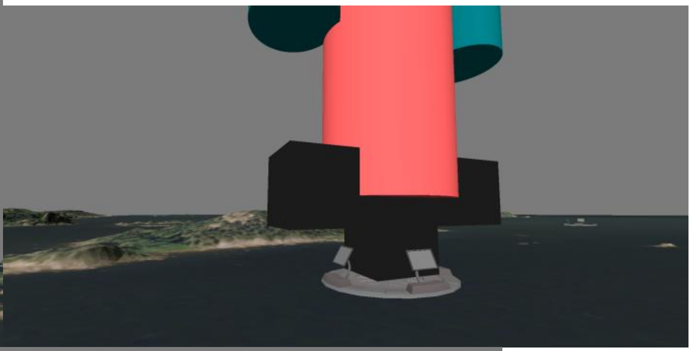
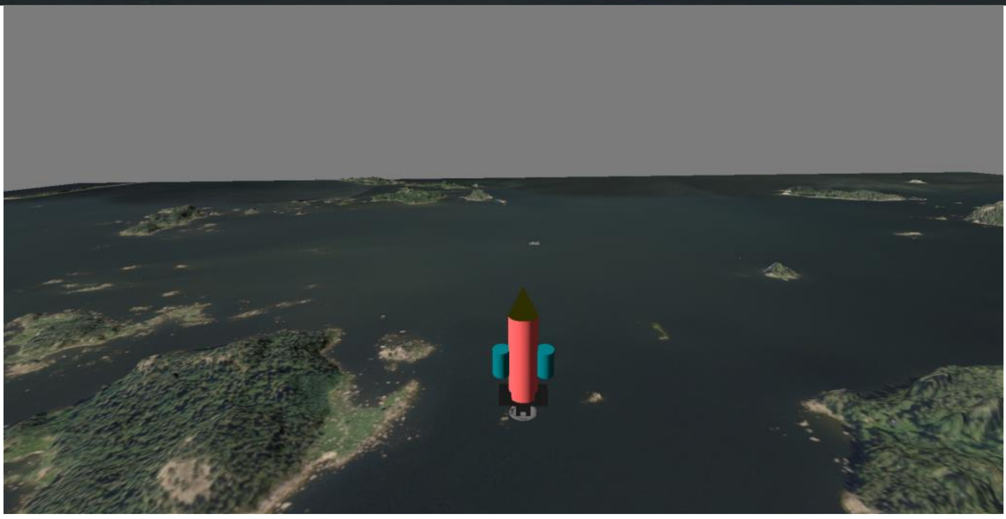
Conclusion
The integration of texture mapping into the OpenGL renderer was successful, enabling the visualisation of orthophotos on the terrain mesh. The renderer now supports detailed texture display, enhancing the visual accuracy of the rendered scene. Future improvements may include the implementation of mipmapping for better texture detail at varying distances and the optimization of texture memory usage.

Task 1.4 Simple Instancing
Scene Setup and Landing Pad Placement
For this scene, two launch pads have been strategically placed within the environment. The followings are the coordinates and orientations for each launch pad:
Launch Pad 1:Position: (x1, y1, z1) = (0.0f, -0.90f, 0.0f); Rotation Angle: angle1 = 0.0 degrees
Launch Pad 2:Position: (x2, y2, z2) = (1.0f, -0.90f, -25.0f); Rotation Angle: angle2 = 45.0 degrees
These coordinates represent the locations of the launch pads in the scene. The positive x-axis points to the right, the positive y-axis points upwards, and the positive z-axis points towards the viewer.
Setting the “white” texture for the launch pads, imitates the setting of the texture of the map: we first load load_texture_2d() and then set it on top of the object: glBindTexture(GL_TEXTURE_2D, whiteTexture); glDrawArrays(GL_TRIANGLES, 0, landingPadVertexCount).



Task 1.5 Custom Model
We have created a 3D rocket(Fig.) made of 3 different shapes(cone, cylinder and cube, all implementations can be found in the respective .cpp/.hpp files).
The rocket has 7 shapes(3 cubes as the base, 3 cylinders and 1 cone). All shapes are connected and have been normalised; we have also added different colours to them.
While merging the shapes I used the concatenate() function.
E.g. auto smallCylinders = concatenate(std::move(smallCylinder1), smallCylinder2);

In the design of the rocket we have used all transformations (translation, rotation, scaling), to first position individually each shape.
E.g.
```
auto smallCylinder1 = make_cylinder(true, 128, { 0.0f, 0.6f, 0.7f },
        make_rotation_z(3.141592f / 2.f)
        * make_scaling(0.9f, 0.3f, 0.4f)
        * make_translation({ 2.f, 2.5f, 0.f }));
```
In order to move the rocket as a whole we merged it with an invisible/mock cube that ties all the shapes together. The rocket is perfectly placed(Fig.) on top of the first launch pad at coordinates (float x1 = 0.0f, y1 = -0.90f, z1 = 0.0f, angle1 = 0.0f;). The mock cube has these exact coordinates placing the centre of the ship on top of the launch pad(* make_translation({ x1, y1, z1 })//platform coords).

Task 1.6 Local Light Sources

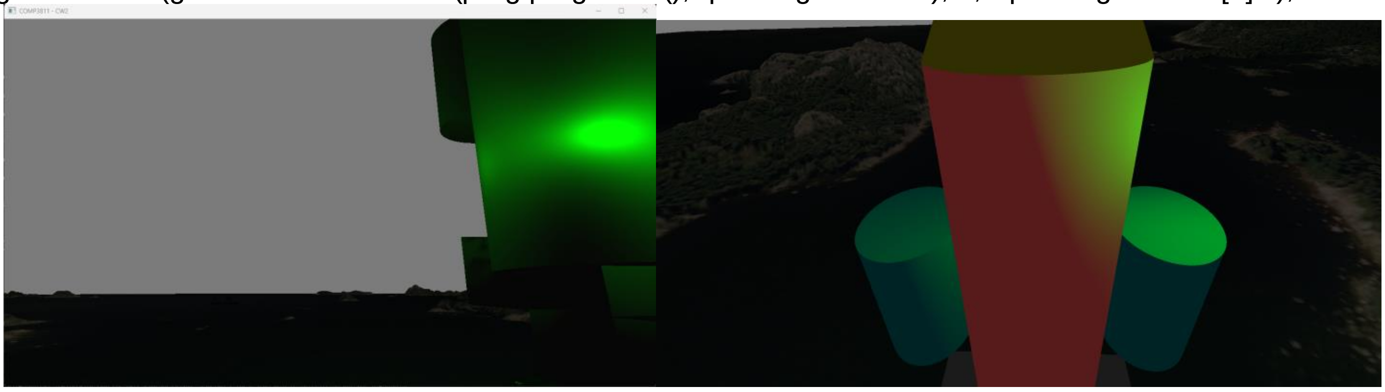
The Blinn-Phong shading model[1] calculates the colour of each pixel based on three components: ambient, diffuse, and specular reflection. The model additionally incorporates distance attenuation for point lighting.

- Ambient Reflection: A constant component that significanty scattering due to the surrounding environmet, rendering its direction undeterminable. Typically, a modest value is assigned to it in order to prevent total darkness.
- Diffuse Reflection: Depends on the angle between the light source and the surface normal. It's brightest when the light is perpendicular to the surface and zero when it's tangential.
- Specular Reflection: Indicates the presence of luminous patches on surfaces that are reflective. It depends on the position of the viewer, the light's orientation, and the shine of the surface.
- Distance Attenuation: For point lights, the intensity of the light diminishes as the distance from the light source increases, typically calculated as $\frac{1}{r^2}$, where r is the distance from the light source.

In .vert we need to transform vertex position to world space and store it in v2fWorldPosition.
```
v2fWorldPosition = (uProjCameraWorld * vec4(iPosition, 1.0)).xyz;
```
In .frag we iterate over the three point lights. We have to set the direction from the current fragment to the point light, calculate the distance between the point light and the fragment, calculate attenuation based on distance, set the diffuse and specular components, combine lighting for the current light, accumulate light contribution, attenuated by distance. "oColor" combines ambient and accumulated light contributions, modulated by texture and vertex colour. This will set the new lighting, in the code we have commented this out, as it doesn't work properly and cancels the general ambient light.
```
    vec3 lightDir = normalize(pointLightPositions[i] - v2fWorldPosition);
    float distance = length(pointLightPositions[i] - v2fWorldPosition);
    float attenuation = 1.0 / (distance * distance);
    vec3 computedIllumination = diff * pointLightColors[i] + spec * pointLightColors[i];
    lightContribution += computedIllumination * attenuation;
oColor = ambient + lightContribution * textureColor * v2fColor;
```

In main.cpp we set the light to point at the spaceship shapes, but unfortunately, the light still follows the camera around. By setting the light position we also set the light colour. And we use glUniform3fv() to display the light from oColor on the map.

```
Mat44f completeShipTransform = make_translation({ completeShipX, completeShipY, completeShipZ });
// Set the uniforms for the point lights
glUniform3fv(glGetUniformLocation(prog.programId(), "pointLightPositions"), 3, &pointLightPositions[0].x);
glUniform3fv(glGetUniformLocation(prog.programId(), "pointLightColors"), 3, &pointLightColors[0].x);
```



Task 1.7 Animation

Introduction

This task focused on implementing a dynamic flight animation for a 3D spaceship model in an OpenGL-based application. The main objective was to create a realistic flight simulation that includes vertical lift-off, curving transition to horizontal flight, and continuous horizontal movement. The implementation was done within the existing OpenGL framework, leveraging the GLFW library for timing and state management.

Implementation Details

The flight animation was incorporated into the application's main loop, utilising a state machine approach to handle different phases of the flight. The states included isLiftingOff, isCurving, and isHorizontalFlight, each dictating the vehicle's behaviour during the respective phases.

Flight Phases

Lift-Off:

The vehicle initially accelerates vertically until it reaches a predefined lift-off height.
The speed of the vehicle gradually increases from an initialSpeed of 0.01f to a maxSpeed of 1.0f, at a rate defined by accelerationRate of 0.001f.

Curving Transition:

After reaching the lift-off height, the vehicle enters a curving motion to transition from vertical to horizontal flight. The curve is defined by a radius (curveRadius) and a maximum curve angle (maxCurveAngle). The vehicle's position and direction are updated based on the elapsed time since the curve started.

Horizontal Flight:

Post-curving, the vehicle transitions to horizontal flight.
The speed continues to increase towards the maximum limit if not already reached.
The vehicle moves forward in the x-direction, simulating a horizontal flight path.

Code Highlights

Speed Calculation: Continual checks ensure that the vehicle's speed does not exceed maxSpeed, incrementing it by accelerationRate until the maximum is reached.

Position and Direction Updates: During the curving phase, trigonometric functions are used to calculate the vehicle's x and y positions, as well as its directional vector.

Challenges and Solutions

Smooth Transition: Ensuring a smooth transition from vertical lift-off to horizontal flight was challenging. This was addressed by carefully calculating the curve parameters and updating the vehicle's position and direction accordingly.

Timing Management: Utilising glfwGetTime() to manage the timing of the animation phases, ensuring frame-rate independence.

Conclusion

The implemented flight animation successfully simulates a realistic spaceship takeoff and transition to horizontal flight. By leveraging GLFW for timing and state management, the application achieves a dynamic and visually appealing animation sequence, adding depth to the 3D rendering capabilities of the OpenGL-based application. This implementation serves as a fundamental step towards creating more complex animations and simulations in the 3D environment.

Task 1.12 Measuring Performance

Introduction

This part of the report presents the findings from the performance measurements conducted on a custom OpenGL rendering application. The primary goal was to analyse the rendering performance, focusing on GPU execution times for different sections of the rendering process.

Methodology

Implementation Details

The performance measurement was integrated into the application's main rendering loop. The following key steps were taken:

Query Object Initialization: OpenGL query objects were created to record timestamps at various points during the rendering process.

Performance Measurement: The rendering times for the entire frame and individual sections (Sections 1.2, 1.4, and 1.5) were measured using glQueryCounter(). Additionally, CPU timing was implemented using std::chrono::high_resolution_clock to measure frame-to-frame intervals and command submission times.

Asynchronous Data Retrieval: To avoid stalling the GPU, an asynchronous approach was used to retrieve timestamp query results at the beginning of each subsequent frame.

Measurement Metrics

The following metrics were recorded:
Full Frame Rendering Time
Rendering Times for Sections 1.2, 1.4, and 1.5
Frame-to-Frame Interval
Command Submission Time

Testing Environment: LINUX Lab machine:
RENDERER llvmpipe (LLVM 16.0.6, 256 bits)
VENDOR Mesa
VERSION 4.5 (Core Profile) Mesa 23.1.4
SHADING_LANGUAGE_VERSION 4.50

Results and Discussion

Task	Metric	System
1.2	GPU time	0.000075ms
1.4	GPU time	680.752694ms
1.5	GPU time	77332.443079ms

Expected Results:

The observed results align with expectations, considering the limitations of the lab machines (slow performance and limited RAM). Task 1.2, involving basic program loading and texture assignment, has a very low GPU time.

As expected, more complex tasks like loading actual textures (Task 1.4) and additional operations (Task 1.5) result in a significant increase in GPU time.

Reproducibility and Variability:

Time varies slightly between reruns based on the position of the camera and how many frames are being loaded. There can be spotted big differences in CPU time for task 1.5. If the spaceship flies away and is no longer in sight, the timing increases, and the closer the spaceship is to the camera the time is lower.

The further away we move from the spaceship the more drastically the task 1.5 increases getting even double the time only by moving away a few metres from the original spot.

Appendix:

	Ayesha	Sandra	Geeyoon
Task 1.1 Matrix/vecto r functions	Created meaningful tests using the Catch2 testing library to validate the implemented functions. Wrote report.		Implemented all the matrix and vector functions: Mat44f operator*(Mat44f const&, Mat44f const&) Vec4f operator*(Mat44f const&, vec4f const&) Mat44f make_rotation_{x,y,z} (float) - Three functions Mat44f make_translation(Vec3f) Mat44f make_perspective_projection(flo at, float, float, float)
1.2 3D renderer basics	Worked on exercises (G1, G2, G3, G4, G5, G6). Focused on shader setup, Mesh Loading and Rendering, general lighting, texture implementation, and camera control debugging for the Coursework. Wrote report.	Worked on the guided exercise (G2,3,4,5,6);Initial texture implementation; Initial camera control.	Worked on the guided exercise (G2,3,4,5,6);Initial texture implementation; Initial camera control.
1.3 Texturing	Worked on implementation of texturing and debugging. Wrote report.	Working on initial implementation of texturing	Working on initial implementation of texturing
1.4 Simple Instancing		Added the white texture to the launch pad. Wrote part of the report.	Placed the launch pad on the map. Fixed positions for both launchpads. Wrote part of the report.
1.5 Custom model	Did initial cylinder implementation. This was used by Sandra to implement the spaceship.	Built the spaceship, implemented a total of 7 basic shapes and 3 different shapes. Placed the spaceship on the launch pad. Wrote report.	
1.6 Local light sources		Tested part of the implementation. Wrote part of the report.	Worked on the initial implementation of Blinn-Phong shading model for point lights. Implemented most of the assignment but since it is no't working, it has been mentioned in the report with implementation trials. Wrote part of the report.
1.7 Animation	Worked on the entire spaceship flight and curved trajectory along with debugging. Wrote report.		

1.8 Tracking cameras	Attempted task. Haven’t been able to implement it correctly yet. Worked together with Sandra to develop a thought process/method to follow.	Attempted task But no successful implementation	
Task 1.12 Measuring Performance	Initial implementation of the code. Wrote theoretical part of the report.	Completed implementation and run task on Linux. Wrote analytic part of the report.	Initial implementation of the code.

Bibliography:

[1]Wikipedia. 2012. Blinn–Phong reflection model. [Online]. [Accessed 13/12/2023]. Available from:https://en.wikipedia.org/wiki/Blinn%E2%80%93Phong_reflection_model