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(π/2)=0 and sin(π/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°)≈1.732051, then v[0]=f/aspect≈0.974279, v[5]=f≈1.732051, v[10]=(far+near)/(near-far)≈-1.002002, v[11]=(2·far·near)/(near-far)≈-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

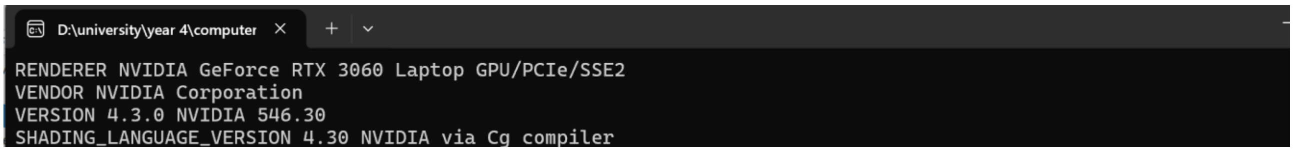
This task implements a basic 3D renderer with camera controls for navigating the terrain model. The implementation includes perspective projection using *make_perspective_projection* function from vmlib, enabling proper 3D depth perception with a 60-degree field of view.

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



```
RENDERER NVIDIA GeForce RTX 3060 Laptop GPU/PCIe/SSE2
VENDOR NVIDIA Corporation
VERSION 4.3.0 NVIDIA 546.30
SHADING_LANGUAGE_VERSION 4.30 NVIDIA via Cg compiler
```

Figure 2.1

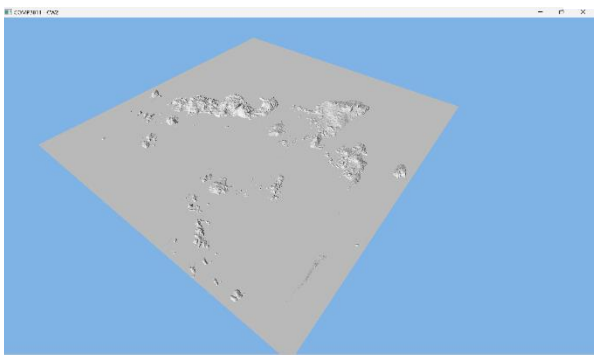


Figure 2.2

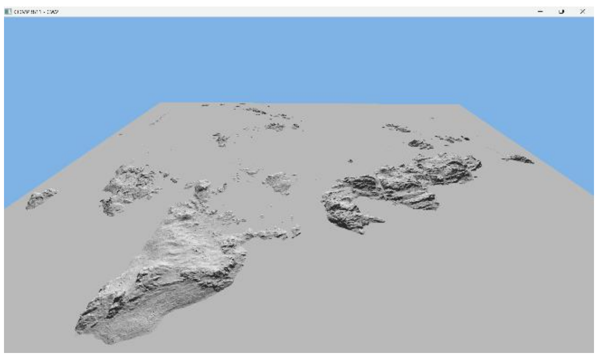


Figure 2.3

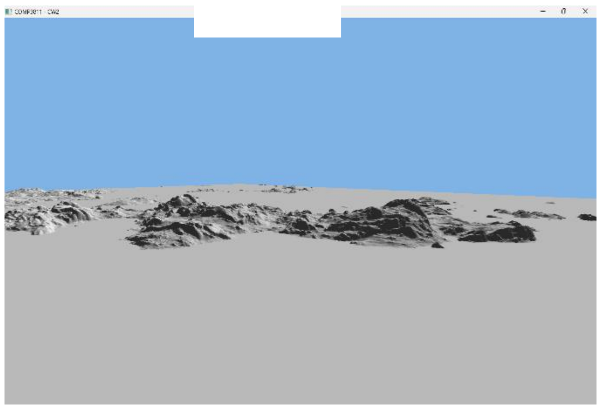


Figure 2.4

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.

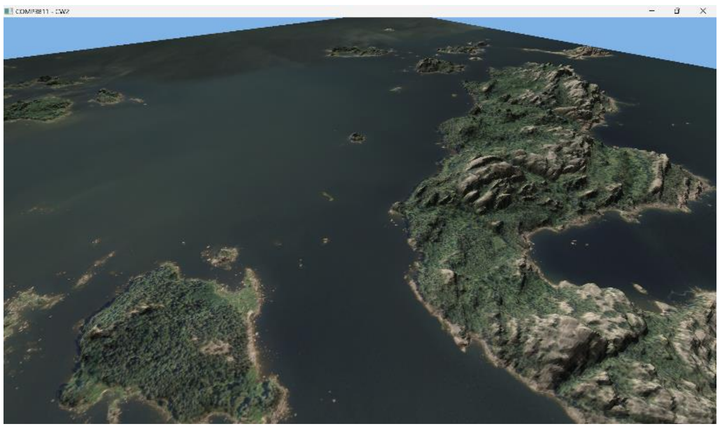


Figure 3.1

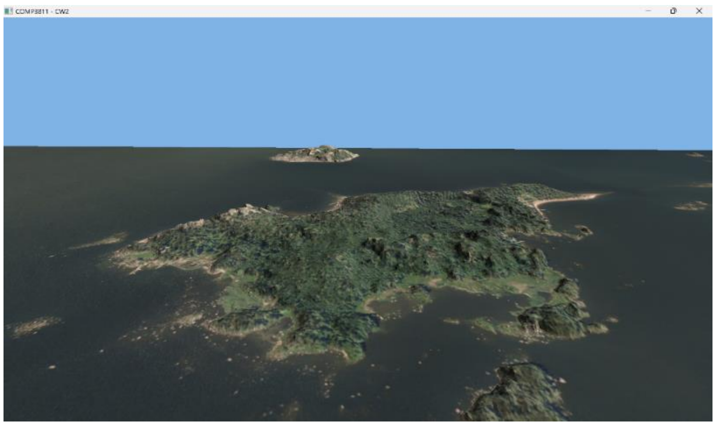


Figure 3.2

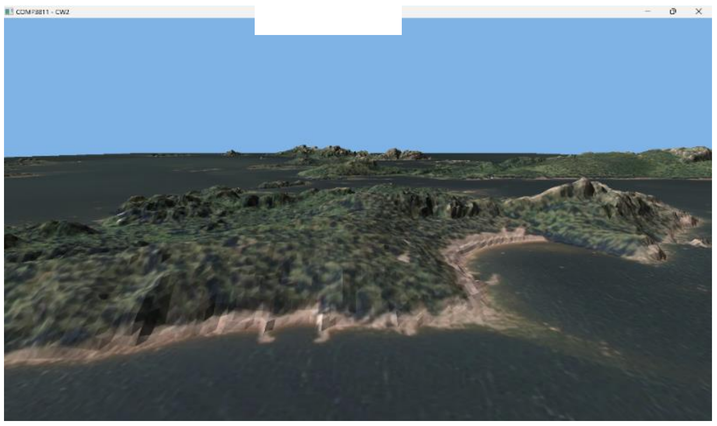


Figure 3.3

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.

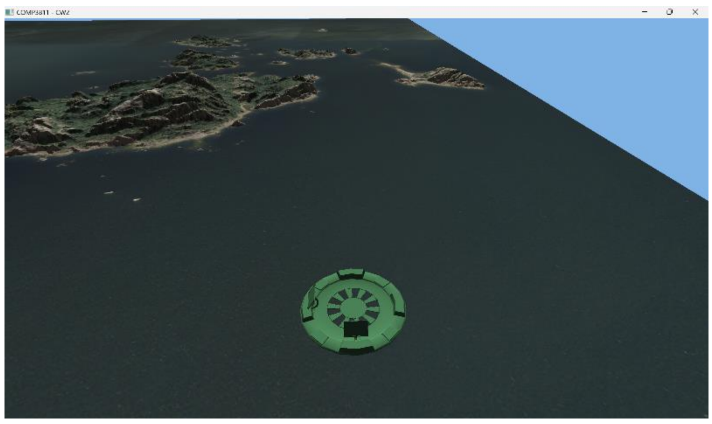


Figure 4.1

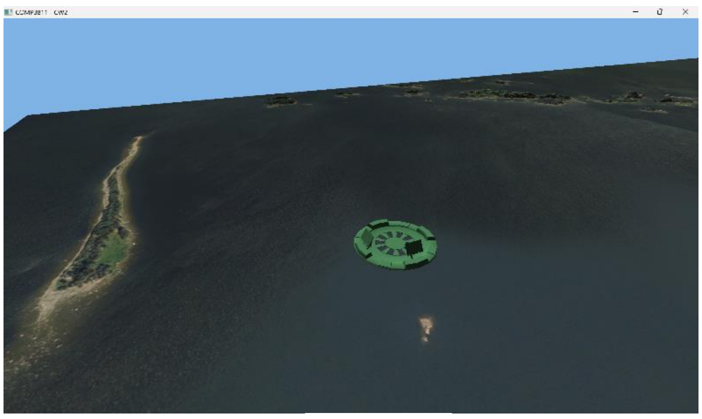


Figure 4.2

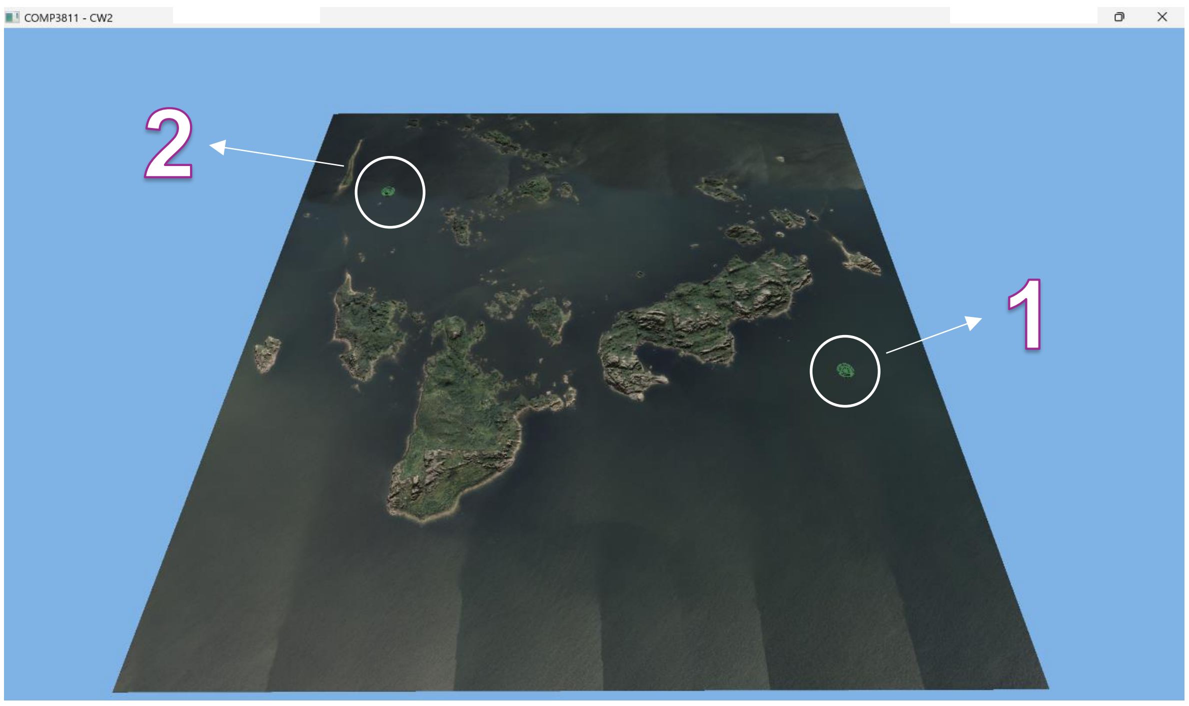


Figure 4.3