Khanh Dong

Linear Algebra

1.

The determinant of the matrix can be calculated as: x0(y1-y2) - x1(y0-y2) + x2(y0-y1)
This represents the area of a parallelogram twice the size of the triangle, which is why we multiply by ½.

2.

Translate p0 to the origin using the translation matrix:

- T-p0 = [1,0,0, -x0], [0,1,0,-y0], [0,0,1,-z0], [0,0,0,1]

The axis direction vector is v = p0-p1
We can normalized to v' = v/||v|| = [v_x, v_y, v_z]

To rotate, we will align the axis to the z coordinate axis, which has two main steps:

- Rotate around the y axis to the xz plane
- Rotate around x axis to align with z axis

To rotate to the xz plane:

- The angle α = arctan2(vx,vz)
- Rotate by -α using the rotation matrix

To align with the z axis:

- Angle β =arctan2(vy,sqrt(vx^2+vz^2))
- Rotate by -β using the rotation matrix

Apply rotation for z-axis using the rotation matrix:

- Rz(θ) = [cos, -sin, 0], [sin, cos, 0], [0,0,1]

Reverse the rotations by rotate around the x-axis by α and then the y-axis by β
Reverse the translation: T-p0 = [1,0,0, x0], [0,1,0,y0], [0,0,1,z0], [0,0,0,1]

The resulting matrix is: M = T-p0 . Ry(-α)Rx(-β)Rz(θ) . Rx(β)Ry(α)Tp0

3.

A rotation in 3D leaves the axis unchanged, and a rotational matrix R is orthogonal (R^T.R = I) with a determinant of 1.
Thus, given the vector v', which is normalized from v=p1-p0, then Rv' = v'.

So the eigenvalue of 1 corresponds to the eigenvector v'. Any distinct point will have an eigenvalue of 1 and its eigenvector along the axis, as long as its p0 != p1.

Report

1 Associate Color with Vertex

The implementation for this can be found in the **helloTriangle.cpp** file. To associate a color with each vertex, the vertex data structure was expanded so that each vertex contains both its position and its color information. This means that for every vertex, the buffer now stores six values: three for the position (x, y, z) and three for the color (r, g, b). The Vertex Buffer Object is filled with this interleaved data, and the Vertex Array Object is configured to interpret the buffer correctly by setting up two attribute pointers, one for position and one for color. The position attribute reads the first three floats of each vertex, while the color attribute reads the next three floats, starting at the appropriate offset in the buffer. This setup ensures that each vertex's color is available to the graphics pipeline during rendering.

The vertex shader was updated to accept both position and color as input attributes. It passes the color attribute to the fragment shader using an output variable. During rasterization, the GPU automatically interpolates the color values across the surface of each triangle, so every fragment receives a smoothly blended color based on the colors of the triangle's vertices. The fragment shader then outputs this interpolated color as the final color of each pixel. The implementation was based on this tutorial.

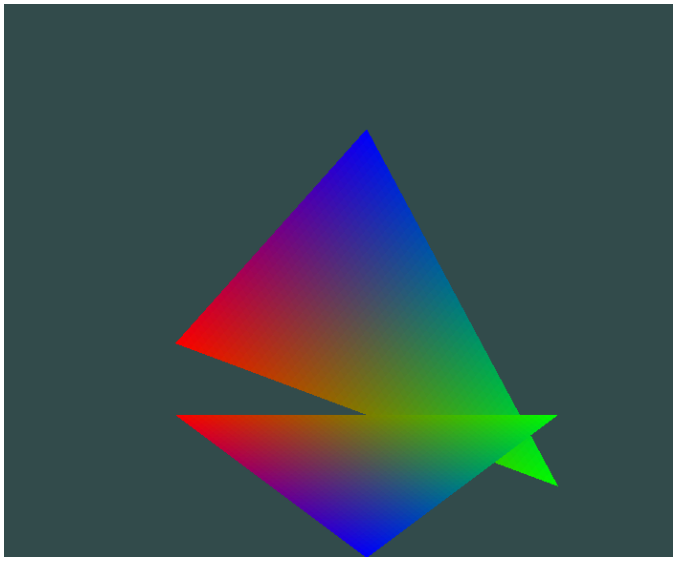

Figure 1. Triangles with color associated with vertex

2 Reading Obj Files

The OBJ file reading functionality was implemented through a parser that processes the ASCII format line by line. When encountering "v" lines, the system stores vertex coordinates in a temporary buffer. For "f" lines indicating faces, the parser extracts vertex indices while handling various format variations. Next is the separate triangles data structure, where each face generates three independent vertices in the final array, creating duplicate vertices even when triangles share geometric points. This transforms indexed geometry into a flat array where every triangle stands alone with its own vertex copies, containing position data (x,y,z) and generated color attributes.

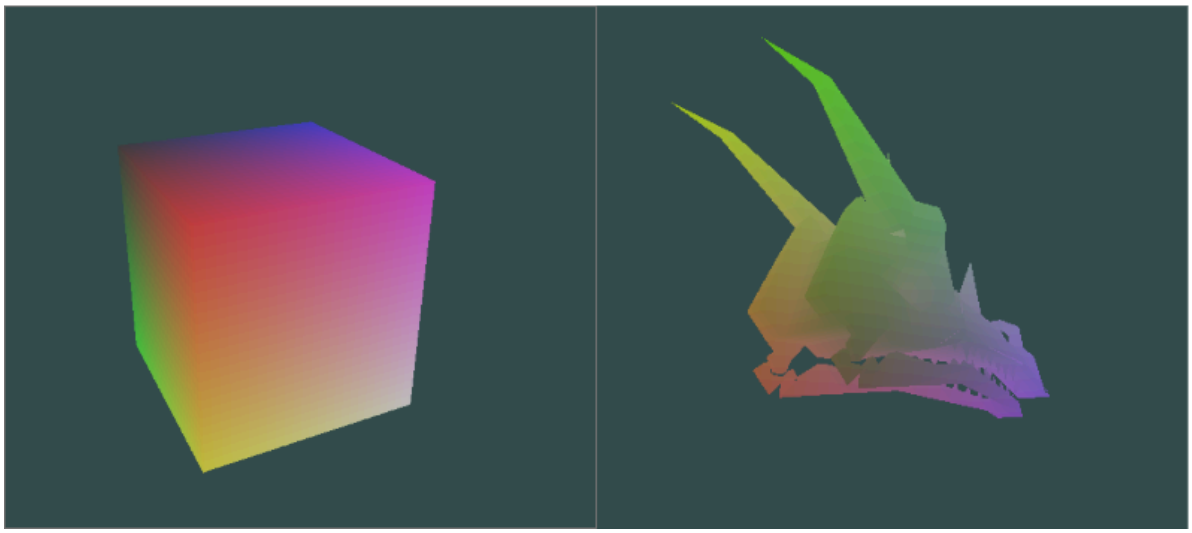

Figure 2. Object rendered with the cube.obj and dragon.obj files

3 Reading Vertex Shader and Fragment Shader files

The shader file reading functionality was implemented by replacing the hardcoded shader strings with a file I/O system. A readShaderFile() function was created that takes a file path as input and returns the shader code as a string.

4 Linear Transformation

Linear transformations were implemented using the GLM library to construct a comprehensive Model-View-Projection matrix system. The implementation created separate transformation matrices for translation, rotation, and scaling, which are combined into a single model matrix using GLM's matrix multiplication functions. The interactive interface was developed using keyboard controls and calling glfwGetKey() to allow real-time manipulation of the object.

For the CPU transformation approach, the system performs all matrix calculations on the CPU side, each vertex position is mathematically transformed by multiplying it with the model matrix using GLM's vector-matrix operations. This occurs in a loop that processes every vertex in the mesh, applying the combined translation, rotation, and scaling transformations. The transformed vertices are then uploaded to the GPU via VBO, and rendering proceeds with simple shaders that pass the pre-transformed coordinates directly to the output.

For the GPU transformation approach, the untransformed vertices remain in GPU memory throughout. The CPU calculates the model, view, and projection matrices using GLM, then sends these matrices to the vertex shader as uniform variables. The actual vertex transformations occur within the vertex shader during the rendering pipeline, where each vertex is transformed by the matrix multiplication gl_Position = projection * view * model * vec4(aPos, 1.0). The implementation is in the **object.cpp** file.

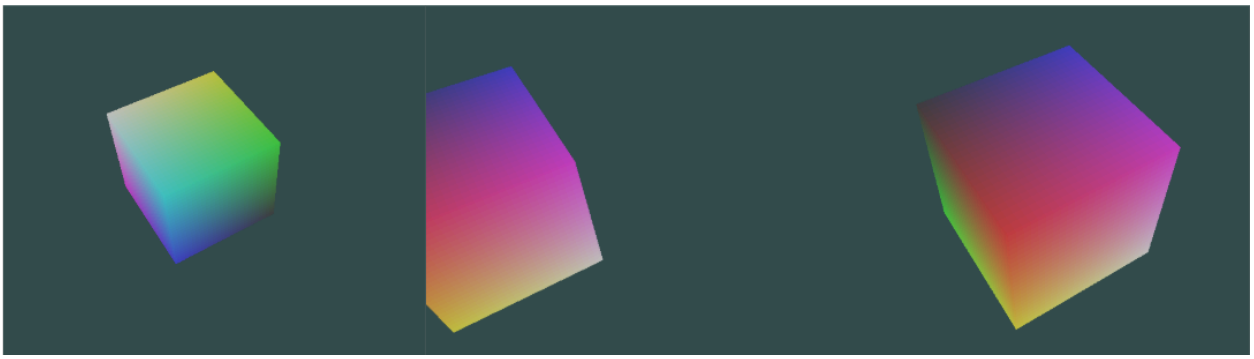

Figure 3. Cube object being applied scaling, translation, and rotation

The file **benchmark.cpp** modified the code to measure performance across different mesh sizes and transformation methods. The implementation used high-resolution timing with std::chrono to capture precise frame times, transformation times, and draw times. A state machine was created to cycle through predefined mesh sizes (100 to 25,000 vertices) and test both CPU and GPU transformation methods for each size. For each test case, the system rendered multiple frames (60) to calculate average performance metrics, eliminating frame-to-frame variance.

The benchmarking process worked by creating test meshes for consistent comparison, then measuring the time taken for vertex transformations (CPU) or matrix uploads (GPU), plus the actual draw call execution time. Results were printed in a structured format tracking frame time, transformation time, draw time, FPS, and memory usage. The result shows that GPU has clear speed advantage and scale effortlessly, while CPU becomes more expensive the more complicated the geometry

```
Detailed Performance Data:
================================================================================
| Vertices | Method | Frame Time | Transform Time | Draw Time | FPS   | Speedup |
================================================================================
|      100 | CPU    |       0.01ms |          0.01ms |     0.00ms | 84507 |     --- |
|      100 | GPU    |       0.00ms |          0.00ms |     0.00ms | 378787 |    4.48x |
|      500 | CPU    |       0.18ms |          0.04ms |     0.14ms | 5465 |     --- |
|      500 | GPU    |       0.00ms |          0.00ms |     0.00ms | 406504 |   74.38x |
|     1000 | CPU    |       0.21ms |          0.09ms |     0.12ms | 4821 |     --- |
|     1000 | GPU    |       0.00ms |          0.00ms |     0.00ms | 369230 |   76.58x |
|     5000 | CPU    |       0.48ms |          0.44ms |     0.04ms | 2087 |     --- |
|     5000 | GPU    |       0.00ms |          0.00ms |     0.00ms | 379506 |  181.76x |
|    10000 | CPU    |       0.98ms |          0.93ms |     0.05ms | 1018 |     --- |
|    10000 | GPU    |       0.00ms |          0.00ms |     0.00ms | 394477 |  387.24x |
|    25000 | CPU    |       2.74ms |          2.64ms |     0.10ms |  364 |     --- |
|    25000 | GPU    |       0.00ms |          0.00ms |     0.00ms | 381194 | 1045.51x |

Performance:
- 100 vertices: CPU=0.01ms, GPU=0.00ms, Speedup=4.48x
- 500 vertices: CPU=0.18ms, GPU=0.00ms, Speedup=74.38x
- 1000 vertices: CPU=0.21ms, GPU=0.00ms, Speedup=76.58x
- 5000 vertices: CPU=0.48ms, GPU=0.00ms, Speedup=181.76x
- 10000 vertices: CPU=0.98ms, GPU=0.00ms, Speedup=387.24x
- 25000 vertices: CPU=2.74ms, GPU=0.00ms, Speedup=1045.51x
```

Figure 4. Benchmarking results

5 Objects Rotation

The rotation implementation used an axis alignment approach where the arbitrary rotation axis between object centers is mathematically transformed to align with the z-axis. This is achieved by calculating the rotation needed to map the custom axis to the z-axis using cross and dot products, then performing a simple rotation around the z-axis, and finally applying the inverse transformation to return to the original orientation. The method creates a composite transformation matrix by aligning the arbitrary axis to z-axis, rotate around z-axis, then realign back to the original coordinate system.

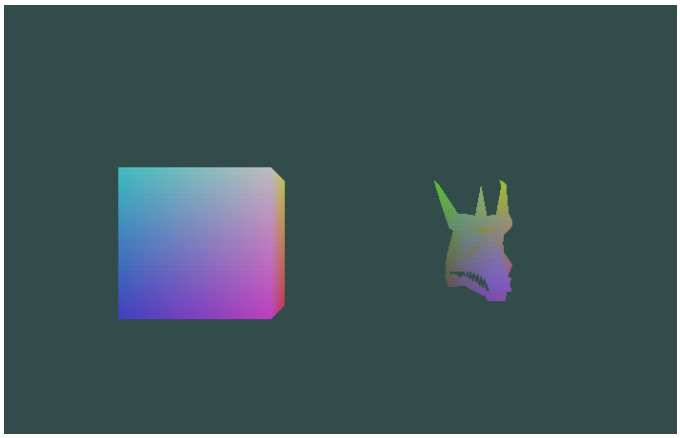

Figure 5. Two objects rendered in the scene