Brock University
Faculty of Mathematics & Science
Department of Computer Science

COSC 3P93: Parallel Computing

4P78 Project Step 2

Prepared By:

Parker TenBroeck 7376726
pt21zs@brocku.ca

Instructor:
Robson De Grande

October 10, 2025

1 Intro

My topic is the rendering of triangle meshes.

This topic is explored in great depth and has a wide array of different techniques and optimizations to produce varying levels of realism. Additionally parallelization often goes hand in hand when talking about these techniques as they are designed for GPUs.

I wanted to implement a set of rendering techniques which offer a good balance between being practical to implement, possible to parallelize, low cost enough to potentially run in real time (30-60 FPS), and be visually interesting. I decided to implement Blinn-Phong shading with the option for ambient/diffuse/specular maps. as well as normal maps. As you will see in some of the demo animations the results are quite visually appealing.

I heavily relied on material and techniques talked about on the Learn OpenGL [1] website.

2 Algorithm Overview

There are a 4 major steps involved in this process

1. Transforming from local space to clip space through the Projection, View, and Model matrices
2. Clipping out of view verticies and triangles.
3. Rasterizing triangles to the frame buffer
4. Performing per fragment(pixel) lighting

2.1 Projection

the projection matrix translates coordinates into clip space (4).

$$\begin{aligned}
fov &= \text{the field of view for our camera} \\
far &= \text{the farthest point our camera can see} \\
near &= \text{the closest point our camera can see} \\
M_{projection} &= \begin{bmatrix} \frac{1}{aspect * \tan(\frac{fov}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{fov}{2})} & 0 & 0 \\ 0 & 0 & -\frac{far+near}{far-near} & -\frac{2 \times far \times near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}
\end{aligned} \tag{1}$$

2.2 View

The view matrix transforms world space coordinates into view space coordinates where x is left and right, y is up and down, and z is depth.

$$\begin{aligned}
P_{target} &= \text{Where the camera is looking} \\
P_{eye} &= \text{Where the camera is} \\
\overrightarrow{V_{world\ up}} &= \text{the up direction of the world } \begin{pmatrix} 0 & 1 & 0 \end{pmatrix} \\
\overrightarrow{V_{cam\ forward}} &= \langle P_{target} - P_{eye} \rangle \\
\overrightarrow{V_{cam\ right}} &= \left\langle \overrightarrow{V_{cam\ forward}} \times \overrightarrow{V_{world\ up}} \right\rangle \\
\overrightarrow{V_{cam\ up}} &= \left\langle \overrightarrow{V_{cam\ right}} \times \overrightarrow{V_{cam\ forward}} \right\rangle \\
M_{view} &= \begin{bmatrix} \overrightarrow{V_{cam\ rightx}} & \overrightarrow{V_{cam\ righty}} & \overrightarrow{V_{cam\ rightz}} & \overrightarrow{V_{cam\ right}} \cdot P_{eye} \\ \overrightarrow{V_{cam\ upx}} & \overrightarrow{V_{cam\ upy}} & \overrightarrow{V_{cam\ upz}} & \overrightarrow{V_{cam\ up}} \cdot P_{eye} \\ -\overrightarrow{V_{cam\ forwardx}} & -\overrightarrow{V_{cam\ forwardy}} & -\overrightarrow{V_{cam\ forwardz}} & \overrightarrow{V_{cam\ forward}} \cdot P_{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{2}$$

2.3 Model

The model matrix transforms coordinates from their local model positions to world space.

$$
\begin{aligned}
&P_{pos} = \text{world space location of the object} \\
&\overrightarrow{V_{scale}} = \text{x, y, z scale of object} \\
&A_{rotation} = \text{euler angles of objects rotation, raw,pitch,roll} \\
&M_{yaw} = \begin{bmatrix} \cos(yaw) & -\sin(yaw) & 0 \\ \sin(yaw) & \cos(yaw) & 0 \\ 0 & 0 & 1 \end{bmatrix} \\
&M_{pitch} = \begin{bmatrix} \cos pitch & 0 & \sin(pitch) \\ 0 & 1 & 0 \\ -\sin pitch & 0 & \cos pitch \end{bmatrix} \\
&M_{roll} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(roll) & -\sin(roll) \\ 0 & \sin(roll) & \cos(roll) \end{bmatrix} \\
&M_{rot} = M_{yaw} \cdot M_{pitch} \cdot M_{roll} \\
&M_{scale} = \begin{bmatrix} \overrightarrow{V_{scalex}} & 0 & 0 \\ 0 & \overrightarrow{V_{scaley}} & 0 \\ 0 & 0 & \overrightarrow{V_{scalez}} \end{bmatrix} \\
&M_{rot\ scale} = M_{rot} \cdot M_{scale} \\
&M_{model} = \begin{bmatrix} M_{rot\ scale 11} & M_{rot\ scale 12} & M_{rot\ scale 13} & P_{pos_x} \\ M_{rot\ scale 21} & M_{rot\ scale 22} & M_{rot\ scale 23} & P_{pos_y} \\ M_{rot\ scale 31} & M_{rot\ scale 32} & M_{rot\ scale 33} & P_{pos_z} \\ 0 & 0 & 0 & 1 \end{bmatrix}
\end{aligned} \tag{3}
$$

2.4 Clip

$$
\begin{aligned}
V_{world} &= M_{model} \cdot V_{local} \\
V_{clip} &= M_{projection} \cdot M_{view} \cdot M_{model} \cdot V_{local} \\
M_{normal} &= {M_{model}^{-1}}^{T} \\
\langle N_{world} \rangle &= M_{normal} \cdot \langle N_{local} \rangle
\end{aligned}
$$

If the resulting vertex is outside this range it is not visible and can be clipped.

$$
\begin{aligned}
-V_{clip_w} \le V_{clip_x} \le V_{clip_w} \\
-V_{clip_w} \le V_{clip_y} \le V_{clip_w} \\
-V_{clip_w} \le V_{clip_z} \le V_{clip_w}
\end{aligned} \tag{4}
$$

2.5 Depth Perspective

After clipping each vertexs x, y, z components are divided by their w component to give depth perspective.

2.6 Screen Space

After depth perspective the x, y components are shifted over by $\frac{1}{2}$ then scaled to the screen width and height.

2.7 Rasterization

The rasterizer first finds a bounding box around the screen space coordinates. Then iterates over each pixel in the box finding the barycentric coordinates at that pixel. It uses these to determine if the point is actually in the triangle or not. and if it is find the liniarly interpreted world position, uv, normal, tangent, and bitangent at that point from the three points defining the triangle.

2.7.1 Perspective Correction

World position, uv, normal, tangent, and bitangent all need to be perspective corrected when being linearly interpolated during rasterization. This is done by scaling each vector type by $\frac{1}{w}$, doing the linear interpolation between face points. then scaling the resulting interpolated vector again by a linearly interpolated $\frac{1}{w}$.

2.8 Fragment

The fragment uses Binn-Phong shading

$$
\begin{aligned}
\left\langle \overrightarrow{V_{light\ dir}} \right\rangle &= \langle P_{light} - P_{fragment} \rangle \\
\text{light power} &= \frac{\text{light intensity}}{|P_{light} - P_{fragment}|^2} \\
\text{lambertian} &= \max(0, \left\langle \overrightarrow{V_{light\ dir}} \right\rangle \cdot \langle N_{fragment} \rangle) \\
\left\langle \overrightarrow{V_{view\ dir}} \right\rangle &= \langle P_{eye} - P_{fragment} \rangle \\
\left\langle \overrightarrow{V_{half\ dir}} \right\rangle &= \left\langle \left\langle \overrightarrow{V_{view\ dir}} \right\rangle + \left\langle \overrightarrow{V_{light\ dir}} \right\rangle \right\rangle \\
\text{specular} &= \max(0, \left\langle \overrightarrow{V_{half\ dir}} \right\rangle \cdot \langle N_{fragment} \rangle)^{\text{fragment shininess}} \\
C_{\text{spec}} &= C_{light} \times \text{specular} \times \text{light power} \\
C_{\text{diff}} &= C_{light} \times \text{lambertian} \times \text{light power}
\end{aligned}
$$

3 Performance

The following tables are the performance tests for 2 different scenes each at 720p and 1080p. The rendered animations can be found here brick.webp and here halo.webp. (please look at the halo one its really cool)

Table 1: Performance Bench Brick 1080p

Resolution	1080x1920
Time	187.95 ms/frame
CPU	AMD Ryzen 7 3800X (16) @ 3.900GHz
OS	Debian GNU/Linux 12 (bookworm) x86_64
RAM	64GiB
Compiler	clang version 14.0.6
Triangles	40337

Table 2: Performance Bench Brick 720p

Resolution	480x720
Time	37.59 ms/frame
CPU	AMD Ryzen 7 3800X (16) @ 3.900GHz
OS	Debian GNU/Linux 12 (bookworm) x86_64
RAM	64GiB
Compiler	clang version 14.0.6
Triangles	40337
Frames	300

Table 3: Performance Bench Halo 1080p

Resolution	1080x1920
Time	71.72 ms/frame
CPU	AMD Ryzen 7 3800X (16) @ 3.900GHz
OS	Debian GNU/Linux 12 (bookworm) x86_64
RAM	64GiB
Compiler	clang version 14.0.6
Triangles	42600
Frames	300

Table 4: Performance Bench Halo 720p

Resolution	480x720
Time	17.62 ms/frame
CPU	AMD Ryzen 7 3800X (16) @ 3.900GHz
OS	Debian GNU/Linux 12 (bookworm) x86_64
RAM	64GiB
Compiler	clang version 14.0.6
Triangles	42600
Frames	300

4 Paralization Opportunity

The code has been setup in a way such that paralization

1. Mesh data is immutable and stored behind a shared pointer allowing access from multiple threads
2. Texture data is immutable and stored behind a shared pointer as well.

3. Scene data is light weight and can be copied with little cost if needed.

   Scene data is the `Camera`, `Light`s, and `Object`s.

   `Object`s only contain a shared pointer to their meshes and material textures so copying them is not costly.

4. When rendering everything aside from the frame buffer and local calculations are immutable and do not depend on eachother for computation.

4.1 Per Frame

The easiest approach is to render each frame in parallel. Because Mesh and Texture data is immutable it can be shared across any number of renderers running in parallel. The only cost would be creating the scene data for each frame running in parallel.

The major downside is each individual frame would still take the same amount of time. This is an issue if we want to use the renderer for real time applications like games.

Additionally each renderer would need its own frame buffer which can be quite large since each fragment(pixel) needs a significant amount of information like ambient, diffuse, specular color and texture data, world position, normal, tangent, bitangent. and more. A 1080x1920 frame buffer is $\sim 250MB$.

4.2 Per Triangle

4.3 Per Fragment (Pixel)

4.4 Combined

Because Triangle

5 Libraries Used

Used `tinyobjloader` [2] for loading and parsing OBJ and MTL files.

Used `stb_image` [3] for loading and parsing texture files into linear RGB.

Used `stb_image_write` [4] for writing rendered frames to disk.

These libraries are only used in the setup of the program when loading the 3D objects and material textures. They have no impact on the rendering performance.

6 Building and Runing

It is Highly reccomended to use Clang when building this project. Clang has much better instruction vectorization optimizations than GCC typically runs 5 times faster (this is only a rough estimate).

Two seperate build files are provided for each respective compiler `build_clang.sh` and `build_gcc.sh`.

Important: to switch between building with gcc or clang you `must` completely delete the build folder if it has been created previously with the other build script.

Once the project has been built you can run it with the `run.sh` script. Which will run the same scene used in 1. 300 frames will be written to `/animation` which can be combined into a animated video with either `make_mp4.sh` or `make_webp.sh` (optional).

You can specify the resolution of the animation by changing the parameters to the program in `run.sh`.

References

[1] “Learn opengl.” [Online]. Available: https://learnopengl.com/

[2] S. Fujita, “Tiny obj loader.” [Online]. Available: https://github.com/tinyobjloader/tinyobjloader

[3] S. Barrett, “stb image.” [Online]. Available: https://github.com/nothings/stb

[4] ——, “stb image write.” [Online]. Available: https://github.com/nothings/stb