Department of Computer Engineering
University of Peradeniya
C Project

CO1020 : Computer Systems Programming

June 01, 2025

Group No: 39
Member 1: E/22/353
Member 2: E/22/184

CONTENT

INTRODUCTION

This is a project report that called **libtiny3d: 3D Software Renderer Library**, where we were built a complete 3D graphics engine from the ground up using 100% of C language.
The project is based on four main tasks.

Our goal was to explore and implement the fundamental concepts behind 3D rendering without relying on any external graphics libraries like OpenGL or DirectX. Instead, we designed and coded every essential component ourselves from pixel manipulation and line drawing to 3D transformations, projection, lighting, and animation.

This project gave us a hands-on opportunity to deeply understand how modern graphics pipelines operate under the hood. We focused on writing efficient mathematical operations, simulating a camera and perspective, rendering 3D wireframes, and adding basic lighting and animation techniques to bring scenes to life. Each task contributed to building a functional and interactive graphics engine capable of producing animated, lit 3D wireframe models on a 2D canvas.

This report documents the design decisions, mathematical foundations, implementation strategies, and challenges we encountered throughout the development of **libtiny3d**.

PROJECT OVERVIEW

The **libtiny3d** project is a lightweight, software-based 3D rendering engine implemented entirely in the C programming language. Unlike traditional graphics engines that rely on hardware acceleration or pre-built libraries like OpenGL, this project demonstrates how 3D scenes can be rendered manually by building every component from scratch.

The development was divided into four key tasks:

- **Task 1: Canvas & Line Drawing**
  Focused on building a floating-point canvas system capable of smooth pixel manipulation and anti-aliased line drawing using the DDA algorithm. The canvas allowed sub-pixel intensity control using bilinear filtering.

- **Task 2: 3D Math Foundation**
  Built a mathematical backbone for the engine, including vector and matrix operations, coordinate transformations, and interpolation techniques like SLERP. These are essential for performing 3D object rotations, translations, and scaling.

- **Task 3: 3D Rendering Pipeline**
  Developed the actual rendering pipeline that transforms 3D vertices through various coordinate systems (Local → World → Camera → Projection → Screen) and renders wireframe models on a 2D canvas. Additional features like viewport clipping and Z-sorting were included.

- **Task 4: Lighting & Animation**
  Added realistic effects and smooth animations. Implemented Lambertian lighting to simulate brightness based on light direction, and cubic Bezier curves to animate 3D object movement over time in a synchronized and looped manner.

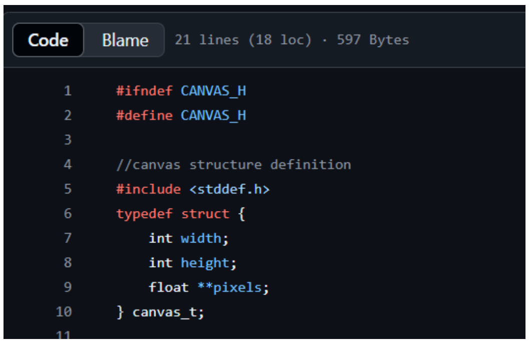
Through this project, we not only applied core systems programming concepts in C, but also integrated mathematical theory, algorithm design, and low-level graphics principles into a cohesive and functional software renderer

Canvas & Line Drawing:

Drawing on the canvas, a 2D surface, is the initial stage in building a 3D renderer. We used sub-pixel accuracy to create a floating-point canvas framework that facilitates smooth, anti-aliased line drawing for this project. Our renderer's visual output layer was generated from this.

Implementation:

- ***canvas_t*** Structure:
  We defined a structure named ***canvas_t*** in ***canvas.h*** file, which holds:
  - ***width*** and ***height*** of the canvas
  - A 2D array of float values between 0.0 and 1.0, representing ***pixel brightness (intensity)***



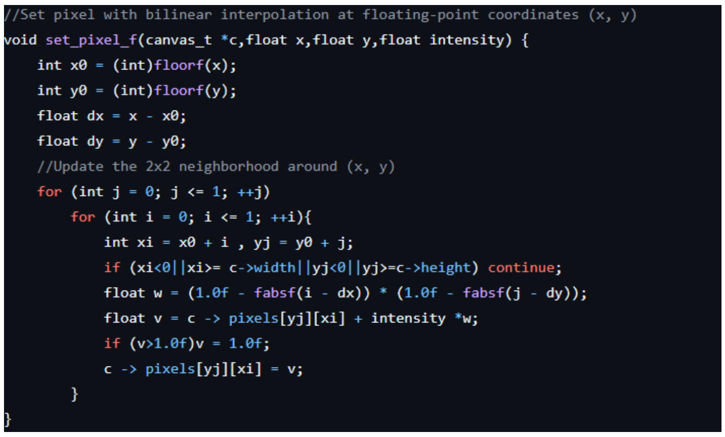
Code Blame 21 lines (18 loc) · 597 Bytes

```c
#ifndef CANVAS_H
#define CANVAS_H

//canvas structure definition
#include <stddef.h>
typedef struct {
    int width;
    int height;
    float **pixels;
} canvas_t;
```

This is our implementing header file which call ***canvas.h***

- **set_pixel_f (canvas, x, y, intensity)**:
  This method, which is implemented in ***canvas.c***, enables us to set a pixel using floating-point coordinates.

  To produce smoother, anti-aliased lines, we use bilinear filtering, which distributes the brightness value proportionately to the four neighboring pixels, as opposed to rounding to the next integer pixel.



```c
//Set pixel with bilinear interpolation at floating-point coordinates (x, y)
void set_pixel_f(canvas_t *c,float x,float y,float intensity) {
    int x0 = (int)floorf(x);
    int y0 = (int)floorf(y);
    float dx = x - x0;
    float dy = y - y0;
    //Update the 2x2 neighborhood around (x, y)
    for (int j = 0; j <= 1; ++j)
        for (int i = 0; i <= 1; ++i){
            int xi = x0 + i , yj = y0 + j;
            if (xi<0||xi>= c->width||yj<0||yj>=c->height) continue;
            float w = (1.0f - fabsf(i - dx)) * (1.0f - fabsf(j - dy));
            float v = c -> pixels[yj][xi] + intensity *w;
            if (v>1.0f)v = 1.0f;
            c -> pixels[yj][xi] = v;
        }
}
```

This is our implementing C file which call ***canvas.c*** in src files location.

Bilinear Filtering:

In computer graphics, bilinear filtering is a texture filtering technique that smoothest out textures whether they are scaled up or down. By combining the colors of the surrounding texels (texture pixels), it lessens blocky, pixelated appearances.

How it works:

- ✓ Locate **the texture coordinate (x, y)** the point on the texture that maps to the screen pixel.
- ✓ **Find the 4 nearest texels** surrounding this point.
- ✓ **Interpolate horizontally** between top texels and between bottom texels (linear interpolation in X).
- ✓ **Interpolate vertically** between the two results above (linear interpolation in Y).

Advantages:

1. Fast and easy to implement

2. Looks much smoother than nearest-neighbor (no blocky pixels)

Disadvantages:

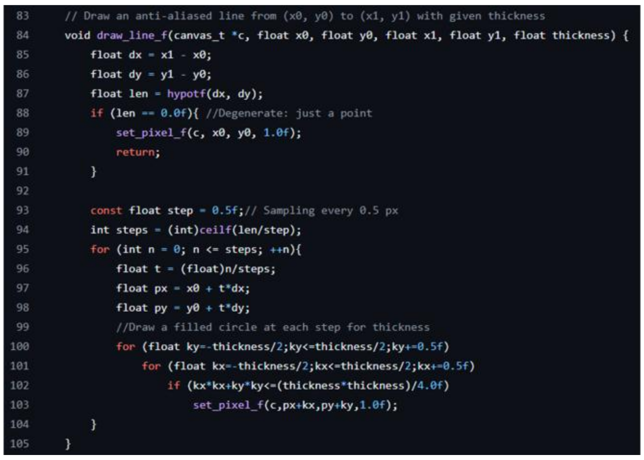
1. Slightly blurry
2. does not support trilinear filtering, anisotropic filtering, sharp edges, or mipmapping.

- **draw_line_f (canvas, x0, y0, x1, y1, thickness):**

  This function creates smooth lines between any two floating-point locations using the **Digital Differential Analyzer (DDA) Algorithm**.

  Additionally, the design allows for variable thickness, so lines can be strong or delicate.

  We used intensity fall-off to blend many parallel lines that were orthogonal to the line's primary direction to manage thickness.



```c
// Draw an anti-aliased line from (x0, y0) to (x1, y1) with given thickness
void draw_line_f(canvas_t *c, float x0, float y0, float x1, float y1, float thickness) {
    float dx = x1 - x0;
    float dy = y1 - y0;
    float len = hypotf(dx, dy);
    if (len == 0.0f){ //Degenerate: just a point
        set_pixel_f(c, x0, y0, 1.0f);
        return;
    }

    const float step = 0.5f;// Sampling every 0.5 px
    int steps = (int)ceilf(len/step);
    for (int n = 0; n <= steps; ++n){
        float t = (float)n/steps;
        float px = x0 + t*dx;
        float py = y0 + t*dy;
        //Draw a filled circle at each step for thickness
        for (float ky=-thickness/2;ky<=thickness/2;ky+=0.5f)
            for (float kx=-thickness/2;kx<=thickness/2;kx+=0.5f)
                if (kx*kx+ky*ky<=(thickness*thickness)/4.0f)
                    set_pixel_f(c,px+kx,py+ky,1.0f);
    }
}
```

DDA Algorithm:

In computer graphics, the DDA algorithm is a line drawing technique used to interpolate data between two locations. On a raster grid (such as pixels on a screen), it determines the intermediate points needed to draw a straight line between a start and an end point.

DDA is based on incremental calculation. It works by calculating either x or y incrementally, depending on the slope, and plotting the closest pixel values.

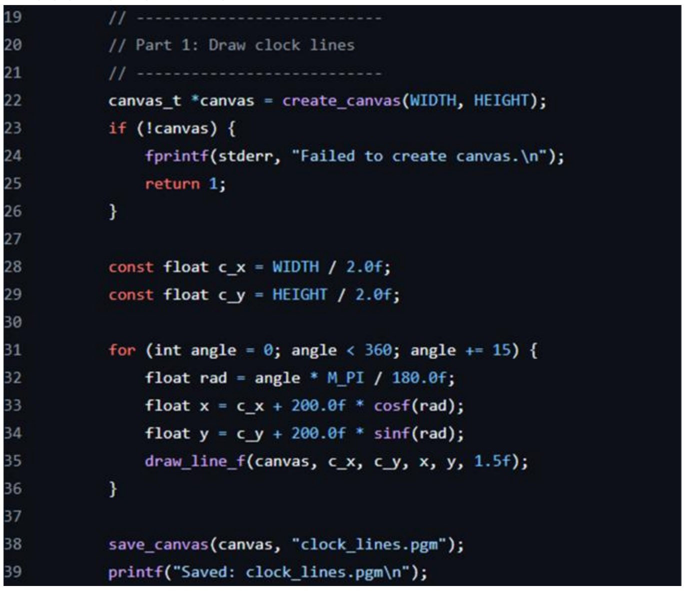
Important Use Cases for project:

- Drawing straight lines in **2D raster graphics.**
- Used in older software renderers or CPU-based rendering systems.

When task 3 of our project builds the entire 3D pipeline on top of pixel level primitives having reliable line primitive early in the pipeline is essential.
So DDA gives us that rock-solid baseline.

Demo:

In the demo program (***main.c***), we created a **radial line pattern** from the center of the canvas, with lines spaced every 15°.
This resembled a clock face and demonstrated the smoothness, accuracy, and anti-aliasing capability of the line-drawing system.
The output clearly showed evenly spaced, smooth lines with variable thickness and brightness transitions.

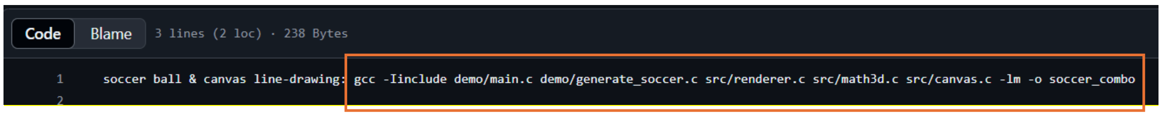


```c
    // --------------------------
    // Part 1: Draw clock lines
    // --------------------------
    canvas_t *canvas = create_canvas(WIDTH, HEIGHT);
    if (!canvas) {
        fprintf(stderr, "Failed to create canvas.\n");
        return 1;
    }

    const float c_x = WIDTH / 2.0f;
    const float c_y = HEIGHT / 2.0f;

    for (int angle = 0; angle < 360; angle += 15) {
        float rad = angle * M_PI / 180.0f;
        float x = c_x + 200.0f * cosf(rad);
        float y = c_y + 200.0f * sinf(rad);
        draw_line_f(canvas, c_x, c_y, x, y, 1.5f);
    }

    save_canvas(canvas, "clock_lines.pgm");
    printf("Saved: clock_lines.pgm\n");
```

This is our implementing C file which call ***main.c*** in demo files location.

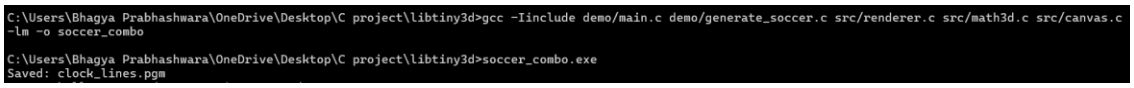
Task 1 run:

This is our run command. When we run this command also task 3 frames & task1 generate. But our required result also given.

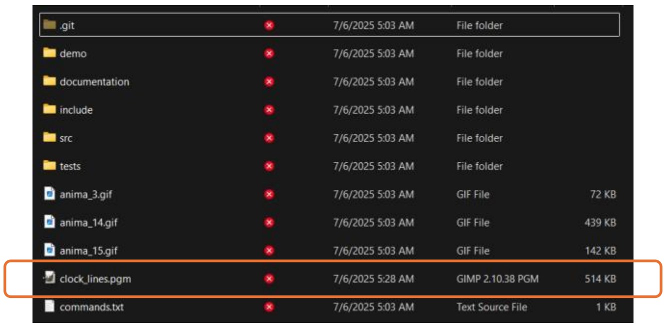


Code Blame 3 lines (2 loc) · 238 Bytes

```
soccer ball & canvas line-drawing: gcc -Iinclude demo/main.c demo/generate_soccer.c src/renderer.c src/math3d.c src/canvas.c -lm -o soccer_combo
```

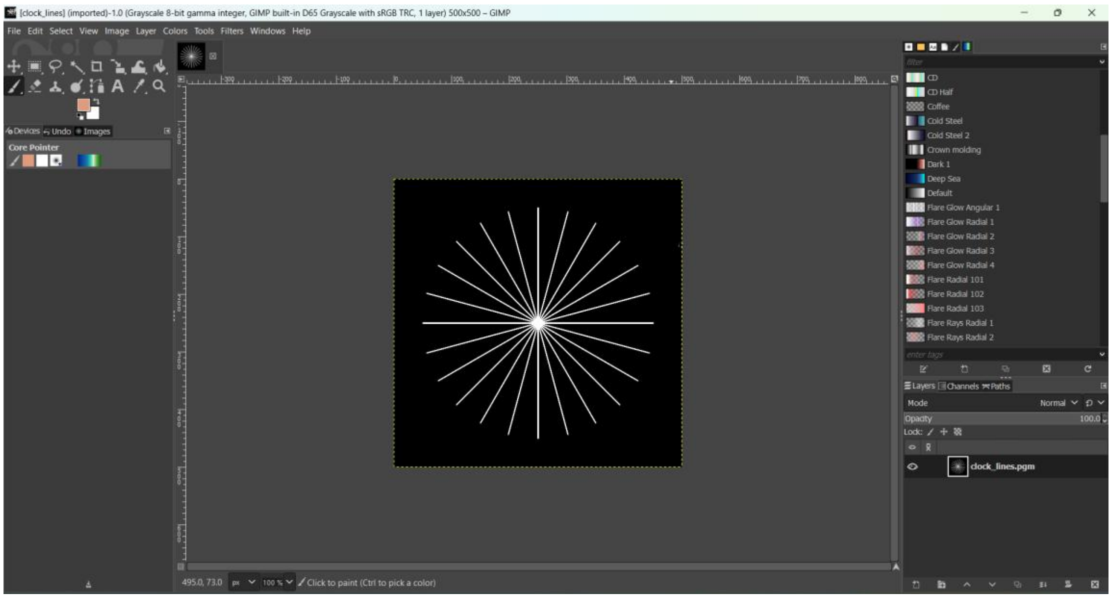


```
C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>gcc -Iinclude demo/main.c demo/generate_soccer.c src/renderer.c src/math3d.c src/canvas.c
-lm -o soccer_combo

C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>soccer_combo.exe
Saved: clock_lines.pgm
```



.git	7/6/2025 5:03 AM	File folder	
demo	7/6/2025 5:03 AM	File folder	
documentation	7/6/2025 5:03 AM	File folder	
include	7/6/2025 5:03 AM	File folder	
src	7/6/2025 5:03 AM	File folder	
tests	7/6/2025 5:03 AM	File folder	
anima_3.gif	7/6/2025 5:03 AM	GIF File	72 KB
anima_14.gif	7/6/2025 5:03 AM	GIF File	439 KB
anima_15.gif	7/6/2025 5:03 AM	GIF File	142 KB
clock_lines.pgm	7/6/2025 5:28 AM	GIMP 2.10.38 PGM	514 KB
commands.txt	7/6/2025 5:03 AM	Text Source File	1 KB

Final output:


[clock_lines] (imported)-1.0 (Grayscale 8-bit gamma integer, GIMP built-in D65 Grayscale with sRGB TRC, 1 layer) 500x500 – GIMP
File Edit Select View Image Layer Colors Tools Filters Windows Help
clock_lines.pgm
100 %
Click to paint (Ctrl to pick a color)

It contains perfect and smooth lines outcome at 100% zoom, initial state. so we could complete task one, correctly.

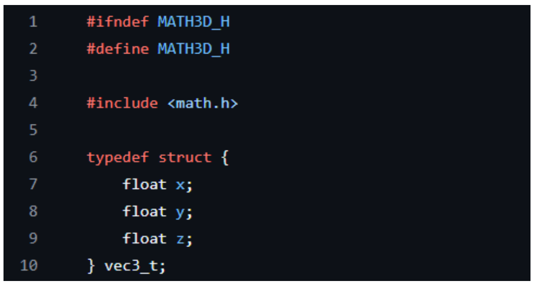
3D Math Foundation:

We constructed the mathematical foundation for object manipulation and 3D transformations in this task.
We created a comprehensive collection of vector and matrix operations from scratch because the project does not rely on third-party libraries for mathematical operations.
To rotate, translate, and scale 3D objects in a scene, these functions are essential.

Implementation:

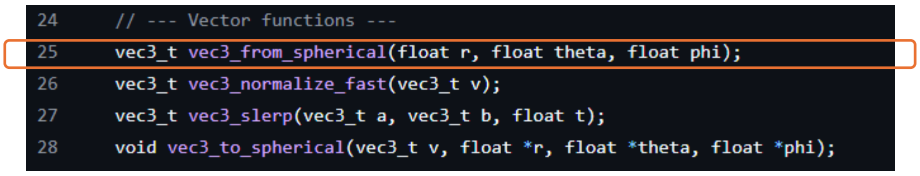
- ***vec3_t*** Structure:
  This structure, defined in ***math3d.h***, uses both to represent a 3D vector:
    1. Coordinates in Cartesian form: x, y, z
    2. Coordinates in a sphere: r, θ, ϕ

  Flexible transformations and interpolations are made possible by the system's automated synchronization of the two representations.



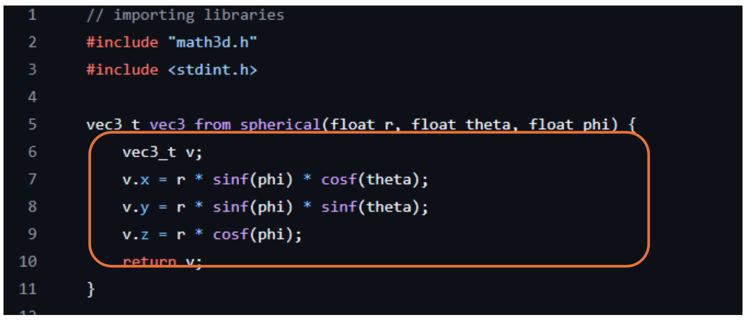
```c
#ifndef MATH3D_H
#define MATH3D_H

#include <math.h>

typedef struct {
    float x;
    float y;
    float z;
} vec3_t;
```

- **vec3_from_spherical(r, theta, phi):**
  Converts spherical coordinates into a Cartesian **vec3_t**. Useful for positioning objects at a given direction and distance from the origin.



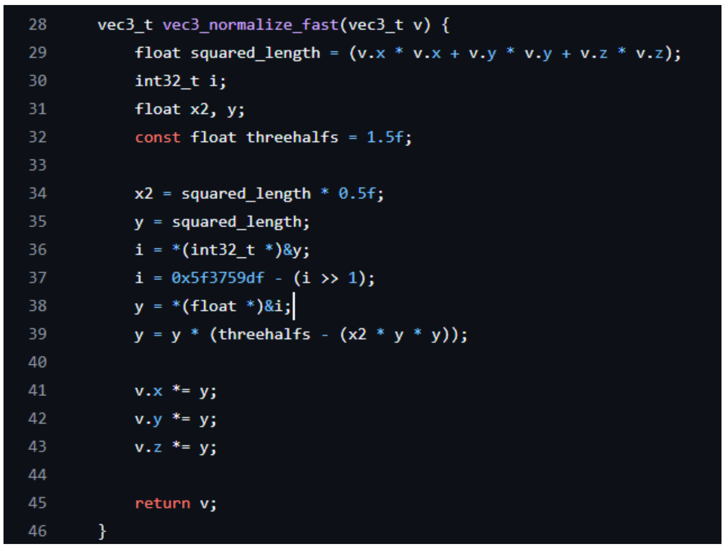
```c
// --- Vector functions ---
vec3_t vec3_from_spherical(float r, float theta, float phi);
vec3_t vec3_normalize_fast(vec3_t v);
vec3_t vec3_slerp(vec3_t a, vec3_t b, float t);
void vec3_to_spherical(vec3_t v, float *r, float *theta, float *phi);
```

Here we coded in inside header file in ***math3d.h*** you can see we use ***vec3_t*** struct, otherwise in ***math3d.c*** file you can look how we handle this vector function for converts spherical coordinates into cartesian



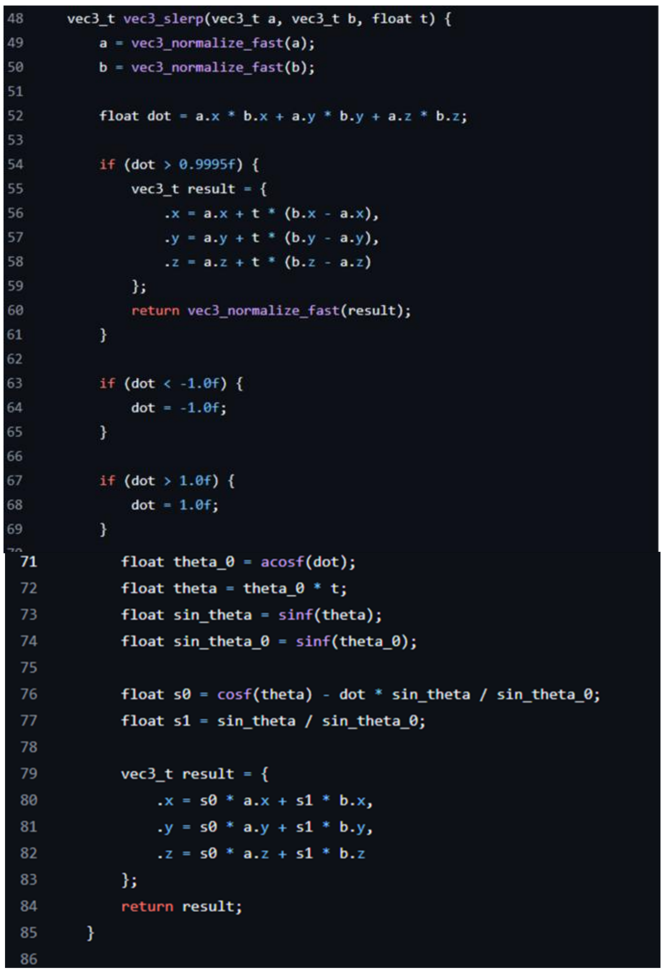
```c
// importing libraries
#include "math3d.h"
#include <stdint.h>

vec3_t vec3_from_spherical(float r, float theta, float phi) {
    vec3_t v;
    v.x = r * sinf(phi) * cosf(theta);
    v.y = r * sinf(phi) * sinf(theta);
    v.z = r * cosf(phi);
    return v;
}
```

- **vec3_normalize_fast():**
  This function normalizes a vector to unit length using an inverse square root trick, which is a well-known optimization in real-time graphics for improving performance while maintaining accuracy.



```c
vec3_t vec3_normalize_fast(vec3_t v) {
    float squared_length = (v.x * v.x + v.y * v.y + v.z * v.z);
    int32_t i;
    float x2, y;
    const float threehalfs = 1.5f;

    x2 = squared_length * 0.5f;
    y = squared_length;
    i = *(int32_t *)&y;
    i = 0x5f3759df - (i >> 1);
    y = *(float *)&i;
    y = y * (threehalfs - (x2 * y * y));

    v.x *= y;
    v.y *= y;
    v.z *= y;

    return v;
}
```

- **vec3_slerp(a, b, t):**
  carries out the two-way Spherical Linear Interpolation (SLERP). This is very helpful for fluid camera transitions and object rotations.



```c
vec3_t vec3_slerp(vec3_t a, vec3_t b, float t) {
    a = vec3_normalize_fast(a);
    b = vec3_normalize_fast(b);

    float dot = a.x * b.x + a.y * b.y + a.z * b.z;

    if (dot > 0.9995f) {
        vec3_t result = {
            .x = a.x + t * (b.x - a.x),
            .y = a.y + t * (b.y - a.y),
            .z = a.z + t * (b.z - a.z)
        };
        return vec3_normalize_fast(result);
    }

    if (dot < -1.0f) {
        dot = -1.0f;
    }

    if (dot > 1.0f) {
        dot = 1.0f;
    }
    float theta_0 = acosf(dot);
    float theta = theta_0 * t;
    float sin_theta = sinf(theta);
    float sin_theta_0 = sinf(theta_0);

    float s0 = cosf(theta) - dot * sin_theta / sin_theta_0;
    float s1 = sin_theta / sin_theta_0;

    vec3_t result = {
        .x = s0 * a.x + s1 * b.x,
        .y = s0 * a.y + s1 * b.y,
        .z = s0 * a.z + s1 * b.z
    };
    return result;
}
```

➢ mat4_t Structure & Matrix Operations:

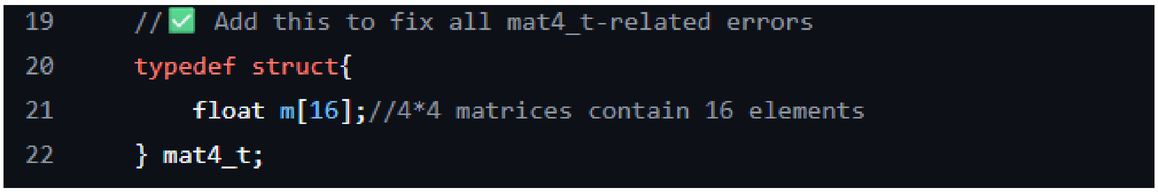
We created a mat4_t structure to handle 4×4 transformation matrices. Internally, matrices are stored in column-major order.

Implemented operations include:

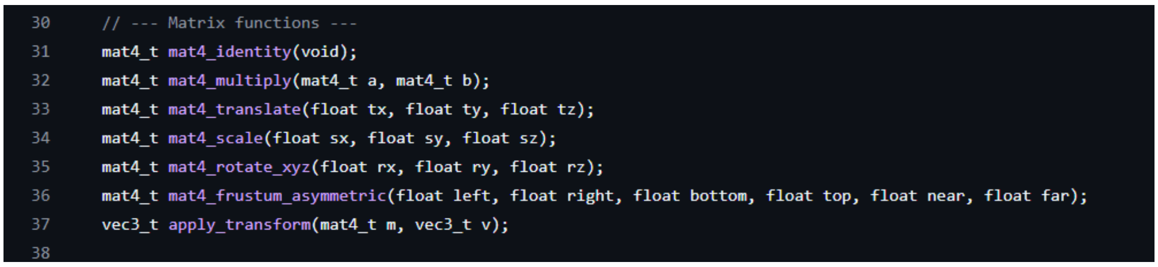
1. *mat4_identity()* = Identity matrix
2. *mat4_translate(tx, ty, tz)* = Translation matrix
3. *mat4_scale(sx, sy, sz)* = Scaling matrix
4. *mat4_rotate_xyz(rx, ry, rz)* = Rotation matrix for all three axes
5. *mat4_frustum_asymmetric(...)* = Asymmetric perspective projection matrix for simulating a camera's field of view

All operations are implemented in ***math3d.c***, with appropriate headers and documentation in ***math3d.h*** file.

In ***math3d.h*** file.

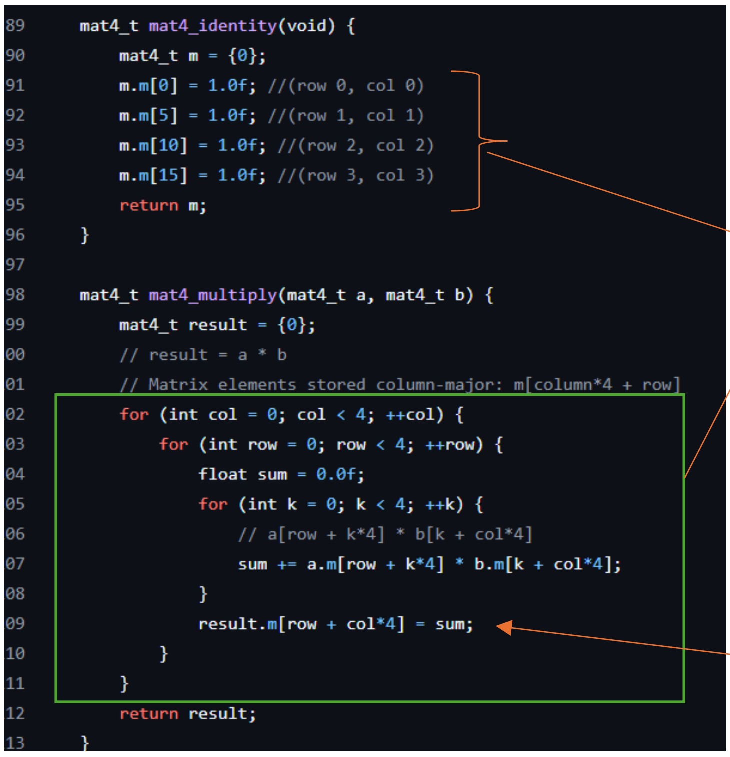


```c
//✅ Add this to fix all mat4_t-related errors
typedef struct{
    float m[16];//4*4 matrices contain 16 elements
} mat4_t;
```



```c
// --- Matrix functions ---
mat4_t mat4_identity(void);
mat4_t mat4_multiply(mat4_t a, mat4_t b);
mat4_t mat4_translate(float tx, float ty, float tz);
mat4_t mat4_scale(float sx, float sy, float sz);
mat4_t mat4_rotate_xyz(float rx, float ry, float rz);
mat4_t mat4_frustum_asymmetric(float left, float right, float bottom, float top, float near, float far);
vec3_t apply_transform(mat4_t m, vec3_t v);

```

How we apply column major order matrices stored:



```c
mat4_t mat4_identity(void) {
    mat4_t m = {0};
    m.m[0] = 1.0f; //(row 0, col 0)
    m.m[5] = 1.0f; //(row 1, col 1)
    m.m[10] = 1.0f; //(row 2, col 2)
    m.m[15] = 1.0f; //(row 3, col 3)
    return m;
}

mat4_t mat4_multiply(mat4_t a, mat4_t b) {
    mat4_t result = {0};
    // result = a * b
    // Matrix elements stored column-major: m[column*4 + row]
    for (int col = 0; col < 4; ++col) {
        for (int row = 0; row < 4; ++row) {
            float sum = 0.0f;
            for (int k = 0; k < 4; ++k) {
                // a[row + k*4] * b[k + col*4]
                sum += a.m[row + k*4] * b.m[k + col*4];
            }
            result.m[row + col*4] = sum;
        }
    }
    return result;
}
```

(0,0) ➔ 0 = 0 + 0*4

(1,1) ➔ 5 = 1 + 1*4

(2,2) ➔ 10 = 2 + 2*4

(3,3) ➔ 15 = 3 + 3*4

//because each column holds 4 rows.

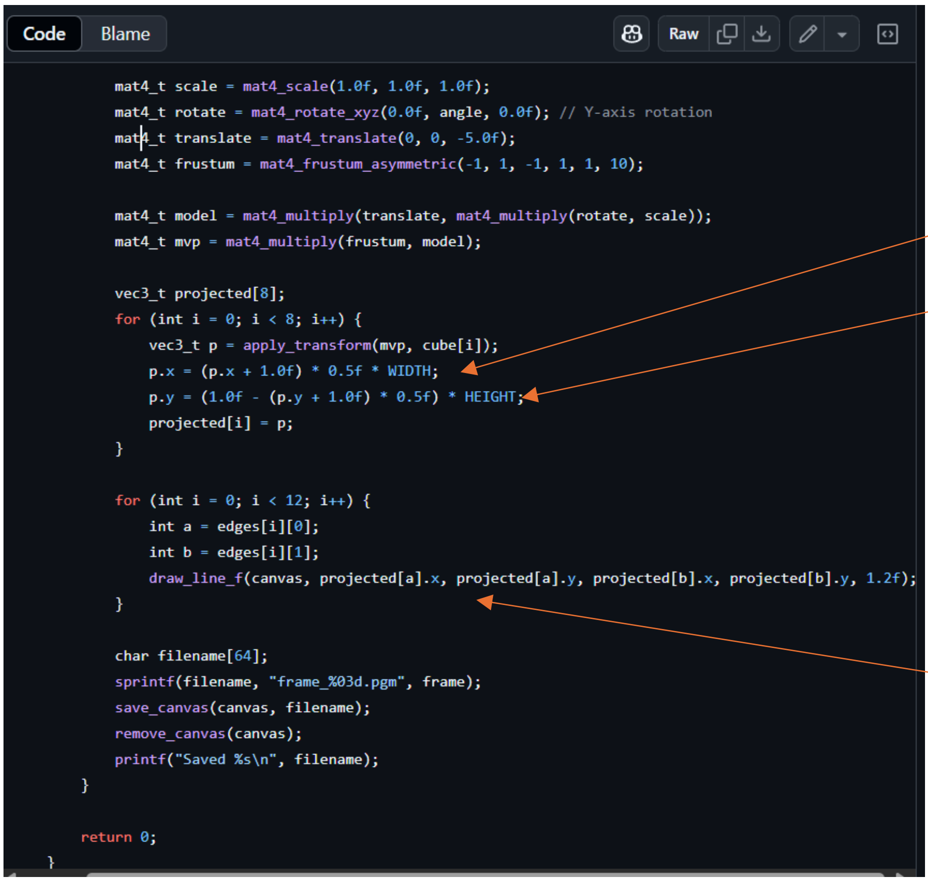
Demo:

To test the math engine, we created a cube in ***test_math.c*** using a set of 3D points. We then applied translation, scaling, and rotation matrices to animate and project the cube into 2D space.

The successful rendering of the transformed cube demonstrated the correctness and flexibility of our math engine.

We use below methods for build the transformation matrices.

mat4_scale(1.0f, 1.0f, 1.0f); = No change, keep the cube's ± 1 size
mat4_rotate_xyz(0.0f, angle, 0.0f) = Spin about **Y** (vertical) axis.
mat4_translate(0, 0, -5.0f); = Push cube 5 units down the Z axis so the camera (at origin) can see it.
mat4_frustum_asymmetric(-1, 1, -1, 1, 1, 10);= Builds a **perspective frustum**: FOV ≈ 90 °, near = 1,far = 10.



```
        mat4_t scale = mat4_scale(1.0f, 1.0f, 1.0f);
        mat4_t rotate = mat4_rotate_xyz(0.0f, angle, 0.0f); // Y-axis rotation
        mat4_t translate = mat4_translate(0, 0, -5.0f);
        mat4_t frustum = mat4_frustum_asymmetric(-1, 1, -1, 1, 1, 10);

        mat4_t model = mat4_multiply(translate, mat4_multiply(rotate, scale));
        mat4_t mvp = mat4_multiply(frustum, model);

        vec3_t projected[8];
        for (int i = 0; i < 8; i++) {
            vec3_t p = apply_transform(mvp, cube[i]);
            p.x = (p.x + 1.0f) * 0.5f * WIDTH;
            p.y = (1.0f - (p.y + 1.0f) * 0.5f) * HEIGHT;
            projected[i] = p;
        }

        for (int i = 0; i < 12; i++) {
            int a = edges[i][0];
            int b = edges[i][1];
            draw_line_f(canvas, projected[a].x, projected[a].y, projected[b].x, projected[b].y, 1.2f);
        }

        char filename[64];
        sprintf(filename, "frame_%03d.pgm", frame);
        save_canvas(canvas, filename);
        remove_canvas(canvas);
        printf("Saved %s\n", filename);
    }

    return 0;
}
```

map x from [−1,1] → [0,WIDTH)

flip Y (top-left origin) and scale

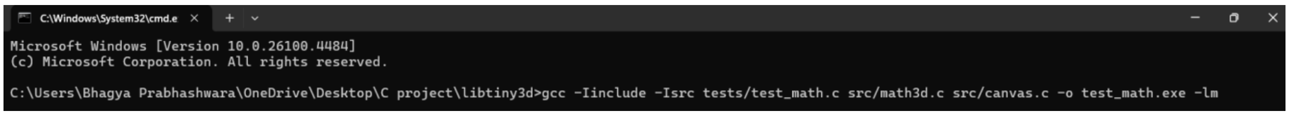
✓ Now *projected[8]* holds **pixel coordinates**.

Draw the cube's 12 edges

✓ The 12 wireframe lines trace a smooth movement since the cube rotates a few degrees per frame.

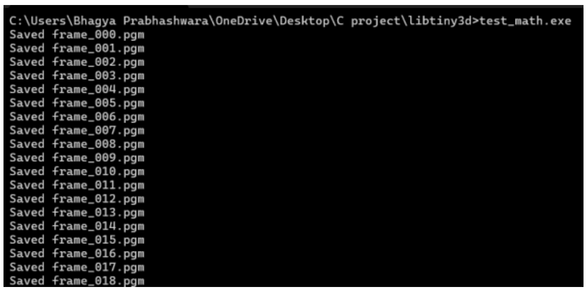
Task 2 run:

This is our run command.

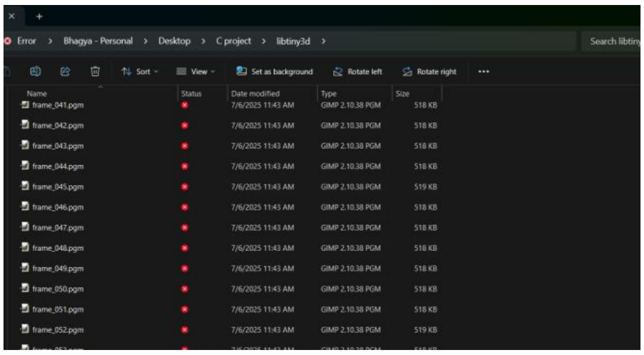


```
Microsoft Windows [Version 10.0.26100.4484]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>gcc -Iinclude -Isrc tests/test_math.c src/math3d.c src/canvas.c -o test_math.exe -lm
```

When we run this command all 60 no: of frames are generated for task 2.



```
C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>test_math.exe
Saved frame_000.pgm
Saved frame_001.pgm
Saved frame_002.pgm
Saved frame_003.pgm
Saved frame_004.pgm
Saved frame_005.pgm
Saved frame_006.pgm
Saved frame_007.pgm
Saved frame_008.pgm
Saved frame_009.pgm
Saved frame_010.pgm
Saved frame_011.pgm
Saved frame_012.pgm
Saved frame_013.pgm
Saved frame_014.pgm
Saved frame_015.pgm
Saved frame_016.pgm
Saved frame_017.pgm
Saved frame_018.pgm
```



Error > Bhagya - Personal > Desktop > C project > libtiny3d >

Name	Status	Date modified	Type	Size
frame_041.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_042.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_043.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_044.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_045.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	519 KB
frame_046.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_047.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_048.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_049.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_050.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_051.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	518 KB
frame_052.pgm	✗	7/6/2025 11:43 AM	GIMP 2.10.38 PGM	519 KB

So, frames are generated, and we implement for 3d animation clip for this all read we uploaded our GitHub repo and, here mention a figure for cube.

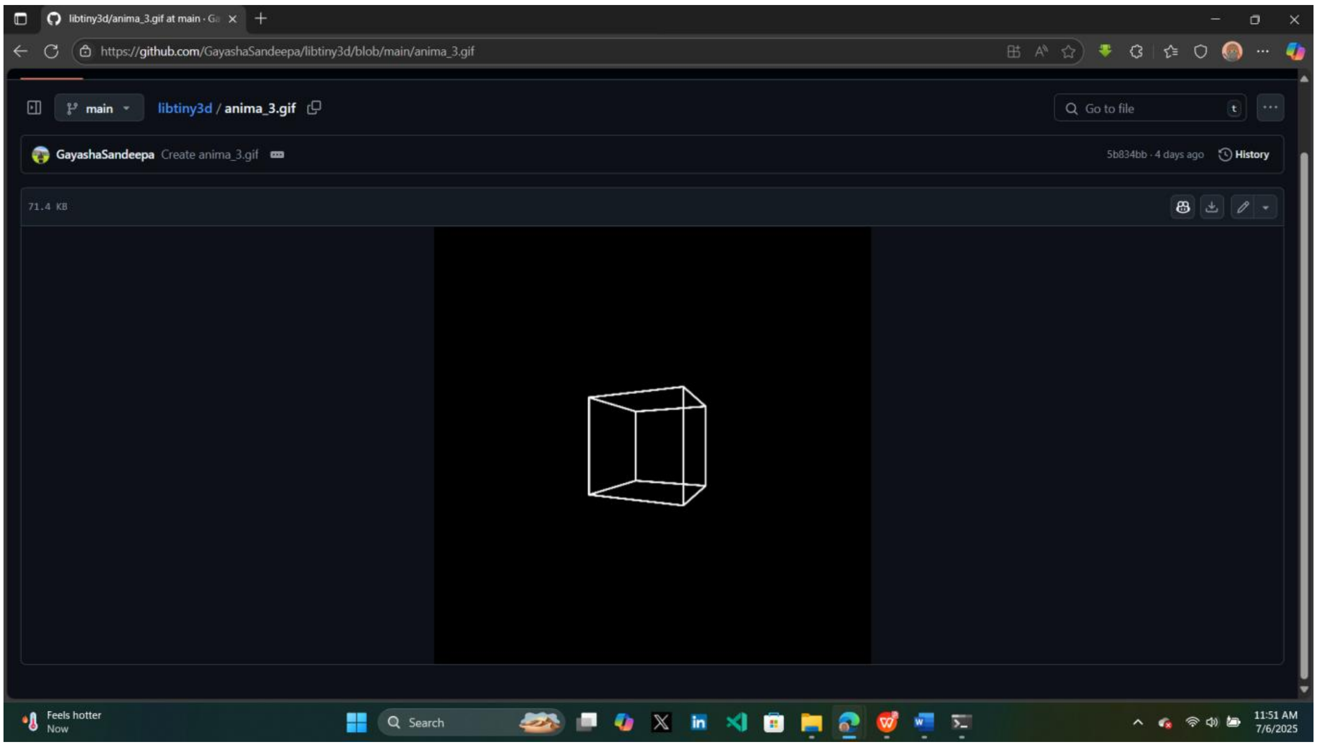

libtiny3d/anima_3.gif at main · Gi
https://github.com/GayashaSandeepa/libtiny3d/blob/main/anima_3.gif
main
libtiny3d / anima_3.gif
Go to file
GayashaSandeepa Create anima_3.gif
5b834bb · 4 days ago
History
71.4 KB
Feels hotter
Now
Search
11:51 AM
7/6/2025

Figure: Rotating cube

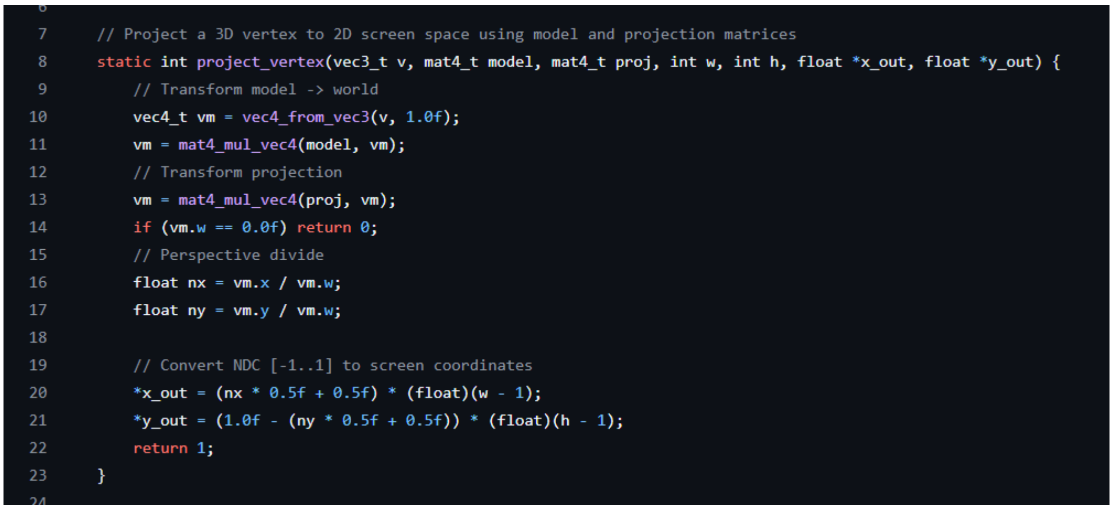
3D Rendering Pipeline:

With the canvas and math foundation in place, this task focused on building the actual 3D rendering pipeline the system that transforms 3D objects into 2D visual representations.
We implemented a pipeline that projects 3D vertices through several transformation stages and renders the output as wireframes on the canvas.

Implementation:

➢ ***project_vertex()***

We implemented in ***renderer.c***, this function takes a 3D point and applies the complete transformation chain.

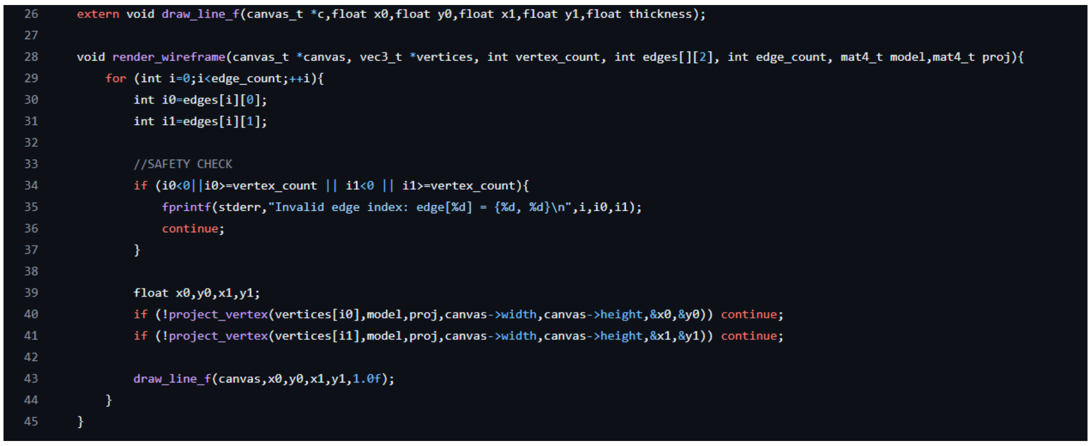


```c
// Project a 3D vertex to 2D screen space using model and projection matrices
static int project_vertex(vec3_t v, mat4_t model, mat4_t proj, int w, int h, float *x_out, float *y_out) {
    // Transform model -> world
    vec4_t vm = vec4_from_vec3(v, 1.0f);
    vm = mat4_mul_vec4(model, vm);
    // Transform projection
    vm = mat4_mul_vec4(proj, vm);
    if (vm.w == 0.0f) return 0;
    // Perspective divide
    float nx = vm.x / vm.w;
    float ny = vm.y / vm.w;

    // Convert NDC [-1..1] to screen coordinates
    *x_out = (nx * 0.5f + 0.5f) * (float)(w - 1);
    *y_out = (1.0f - (ny * 0.5f + 0.5f)) * (float)(h - 1);
    return 1;
}
```

Each step uses matrix multiplications from ***math3d.c*** to simulate the object's orientation, camera position, and final screen coordinates.

➢ ***render_wireframe()***

This is a heart of the pipeline. This function connects transformed vertices using ***draw_line_f()*** to render 3D shapes as **wireframe models**.

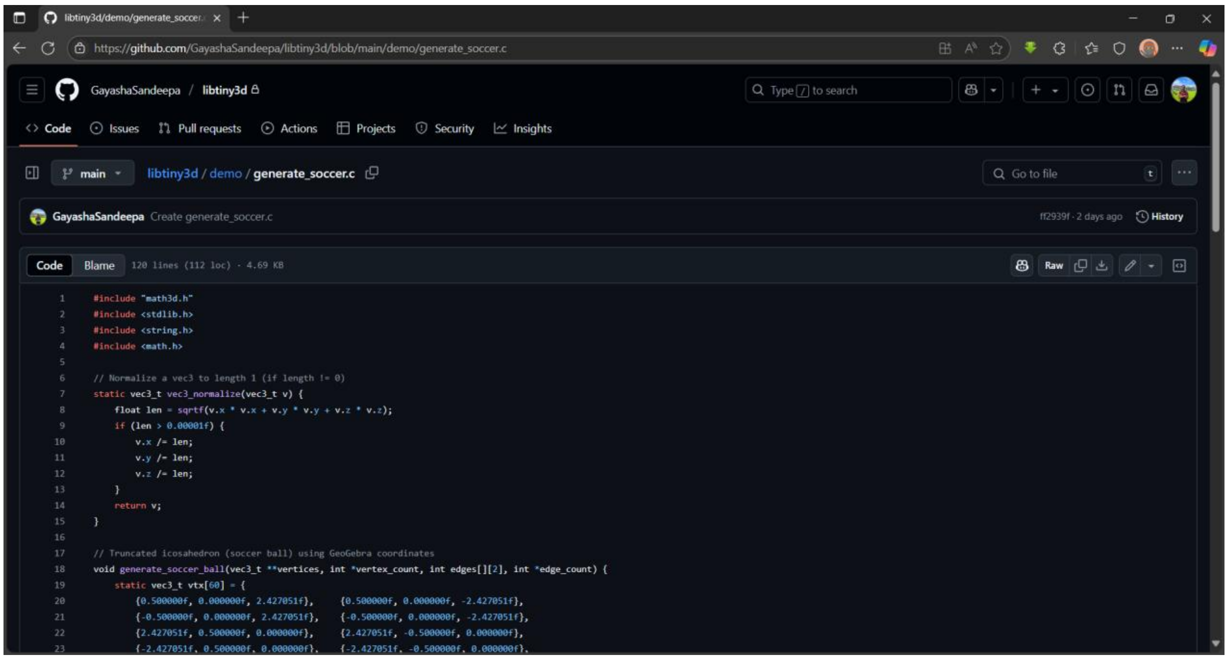


```c
extern void draw_line_f(canvas_t *c,float x0,float y0,float x1,float y1,float thickness);

void render_wireframe(canvas_t *canvas, vec3_t *vertices, int vertex_count, int edges[][2], int edge_count, mat4_t model,mat4_t proj){
    for (int i=0;i<edge_count;++i){
        int i0=edges[i][0];
        int i1=edges[i][1];

        //SAFETY CHECK
        if (i0<0||i0>=vertex_count || i1<0 || i1>=vertex_count){
            fprintf(stderr,"Invalid edge index: edge[%d] = {%d, %d}\n",i,i0,i1);
            continue;
        }

        float x0,y0,x1,y1;
        if (!project_vertex(vertices[i0],model,proj,canvas->width,canvas->height,&x0,&y0)) continue;
        if (!project_vertex(vertices[i1],model,proj,canvas->width,canvas->height,&x1,&y1)) continue;

        draw_line_f(canvas,x0,y0,x1,y1,1.0f);
    }
}
```

- ***generate_soccer_ball()***

We not it adds for the ***main.c*** file, we got helper file option, and it create ***generate_soccer.c*** inside demo.
To showcase our renderer, we generated a **truncated icosahedron** (the geometric structure of a soccer ball).
This involved:

1. Creating the vertex positions manually or through algorithmic construction. We got 60 vertex points
2. Defining connectivity (edges).
3. Passing the data to ***render_wireframe()*** for visualization



https://github.com/GayashaSandeepa/libtiny3d/blob/main/demo/generate_soccer.c

libtiny3d / demo / generate_soccer.c

GayashaSandeepa Create generate_soccer.c — ff2939f · 2 days ago

120 lines (112 loc) · 4.69 KB

```c
#include "math3d.h"
#include <stdlib.h>
#include <string.h>
#include <math.h>

// Normalize a vec3 to length 1 (if length != 0)
static vec3_t vec3_normalize(vec3_t v) {
    float len = sqrtf(v.x * v.x + v.y * v.y + v.z * v.z);
    if (len > 0.00001f) {
        v.x /= len;
        v.y /= len;
        v.z /= len;
    }
    return v;
}

// Truncated icosahedron (soccer ball) using GeoGebra coordinates
void generate_soccer_ball(vec3_t **vertices, int *vertex_count, int edges[][2], int *edge_count) {
    static vec3_t vtx[60] = {
        {0.500000f, 0.000000f, 2.427051f},    {0.500000f, 0.000000f, -2.427051f},
        {-0.500000f, 0.000000f, 2.427051f},   {-0.500000f, 0.000000f, -2.427051f},
        {2.427051f, 0.500000f, 0.000000f},    {2.427051f, -0.500000f, 0.000000f},
        {-2.427051f, 0.500000f, 0.000000f},   {-2.427051f, -0.500000f, 0.000000f},
```

Here clearly show our implementing helper file.

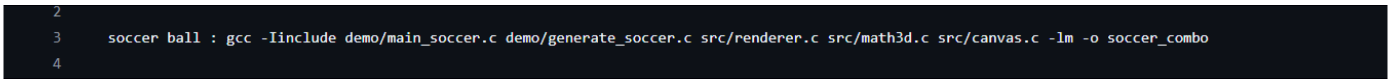
Demo:

In the demo application (***generate_soccer.c***), we rendered a rotating soccer ball model inside a circular viewport. The ball was animated to rotate in real time, clearly showing 3D depth, smooth transformation, and wireframe rendering.

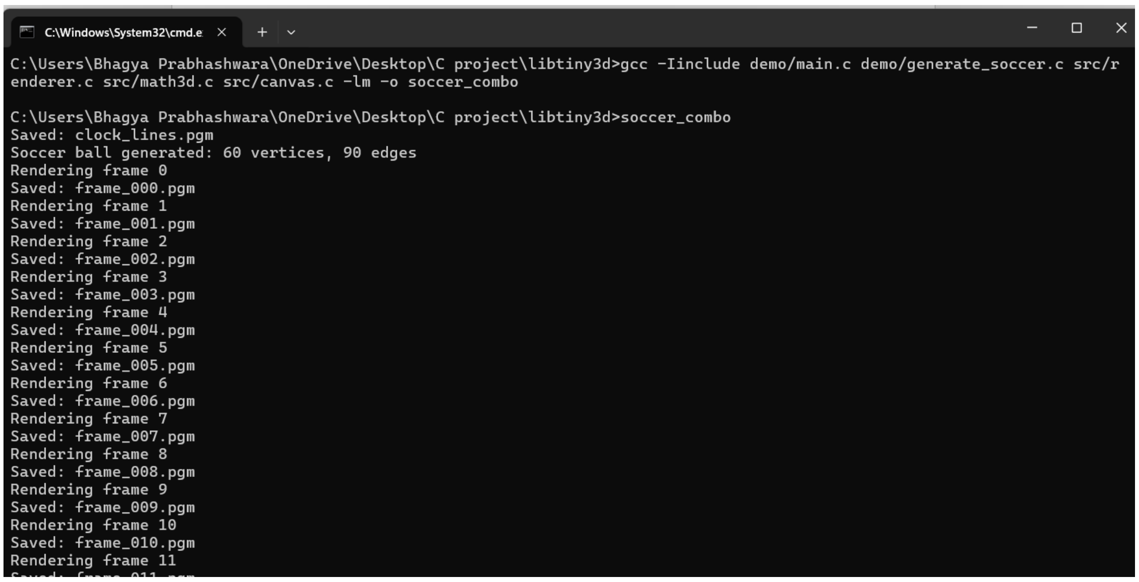
The result was a clean and responsive 3D visualization pipeline capable of real-time rendering with pure software logic.

Task 3 run:

This is our cmd run command , and it gives 60 frames for the generate for rendering to achieve final output.

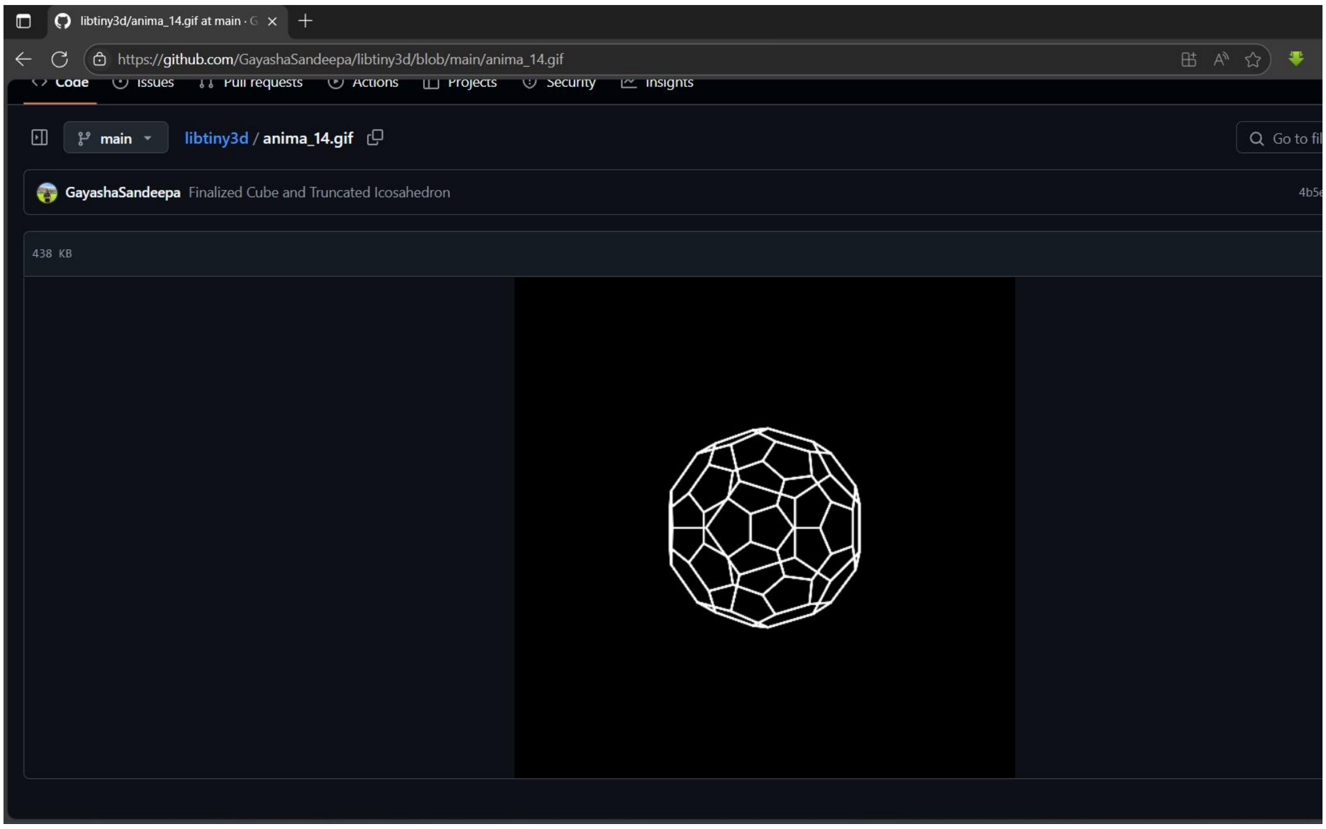


```
soccer ball : gcc -Iinclude demo/main_soccer.c demo/generate_soccer.c src/renderer.c src/math3d.c src/canvas.c -lm -o soccer_combo
```



```
C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>gcc -Iinclude demo/main.c demo/generate_soccer.c src/r
enderer.c src/math3d.c src/canvas.c -lm -o soccer_combo

C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>soccer_combo
Saved: clock_lines.pgm
Soccer ball generated: 60 vertices, 90 edges
Rendering frame 0
Saved: frame_000.pgm
Rendering frame 1
Saved: frame_001.pgm
Rendering frame 2
Saved: frame_002.pgm
Rendering frame 3
Saved: frame_003.pgm
Rendering frame 4
Saved: frame_004.pgm
Rendering frame 5
Saved: frame_005.pgm
Rendering frame 6
Saved: frame_006.pgm
Rendering frame 7
Saved: frame_007.pgm
Rendering frame 8
Saved: frame_008.pgm
Rendering frame 9
Saved: frame_009.pgm
Rendering frame 10
Saved: frame_010.pgm
Rendering frame 11
Saved: frame_011.pgm
```

This is our implemention : Rotating **truncated icosahedron**


libtiny3d/anima_14.gif at main · G
https://github.com/GayashaSandeepa/libtiny3d/blob/main/anima_14.gif
Code
Issues
Pull requests
Actions
Projects
Security
Insights
main
libtiny3d / anima_14.gif
Go to fi
GayashaSandeepa Finalized Cube and Truncated Icosahedron
4b5e
438 KB

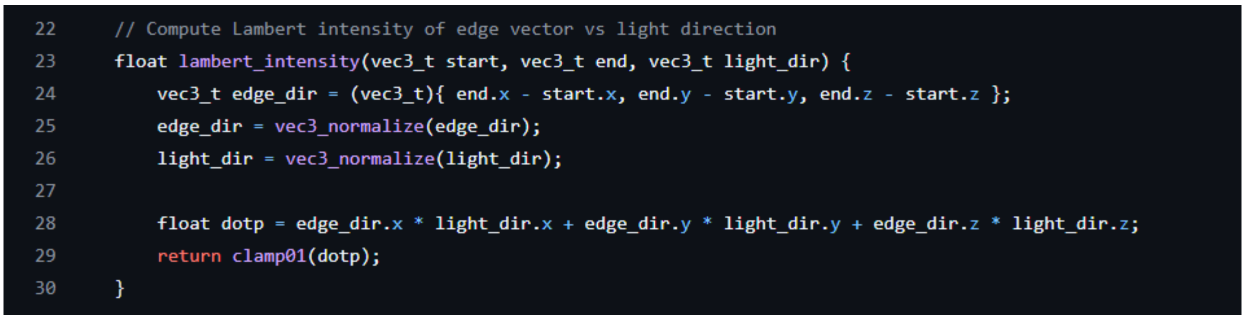
Lighting & Animation:

The final stage of the project focused on enhancing visual realism and dynamics. By introducing **lighting** and **animation**, we elevated the renderer from a static wireframe viewer to a dynamic visual engine capable of producing more lifelike and engaging scenes.

Implementation:

➢ ***Lambert Lighting Model:***

Implemented in ***lighting.c***, we used the **Lambertian reflectance model** to calculate the brightness of each line segment based on the angle between the line's direction and the incoming light direction.

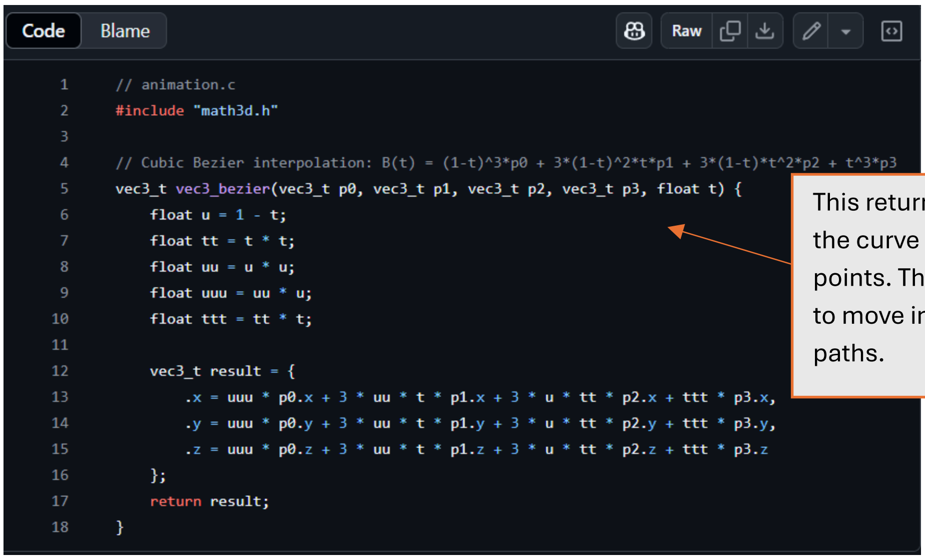


```c
// Compute Lambert intensity of edge vector vs light direction
float lambert_intensity(vec3_t start, vec3_t end, vec3_t light_dir) {
    vec3_t edge_dir = (vec3_t){ end.x - start.x, end.y - start.y, end.z - start.z };
    edge_dir = vec3_normalize(edge_dir);
    light_dir = vec3_normalize(light_dir);

    float dotp = edge_dir.x * light_dir.x + edge_dir.y * light_dir.y + edge_dir.z * light_dir.z;
    return clamp01(dotp);
}
```

Above our code implementation represent the how we implement the Lambertian reflectance model. We extended the system to support light source, combining their contributions for more realistic shading.

➢ ***Bézier Curve Animation:***

Implemented in ***animation.c***, we used cubic Bézier curves to smoothly animate objects in 3D space.



```c
// animation.c
#include "math3d.h"

// Cubic Bezier interpolation: B(t) = (1-t)^3*p0 + 3*(1-t)^2*t*p1 + 3*(1-t)*t^2*p2 + t^3*p3
vec3_t vec3_bezier(vec3_t p0, vec3_t p1, vec3_t p2, vec3_t p3, float t) {
    float u = 1 - t;
    float tt = t * t;
    float uu = u * u;
    float uuu = uu * u;
    float ttt = tt * t;

    vec3_t result = {
        .x = uuu * p0.x + 3 * uu * t * p1.x + 3 * u * tt * p2.x + ttt * p3.x,
        .y = uuu * p0.y + 3 * uu * t * p1.y + 3 * u * tt * p2.y + ttt * p3.y,
        .z = uuu * p0.z + 3 * uu * t * p1.z + 3 * u * tt * p2.z + ttt * p3.z
    };
    return result;
}
```

This returns a position at time float t on the curve defined by four control points. This technique allows objects to move in smooth, natural-looking paths.

- ***Synchronized Multi-Object Animation:***
  Multiple wireframe objects were animated simultaneously along Bézier paths. We used the same global time step to keep them in sync. Additionally, we ensured that animations looped smoothly (start position equals end position), creating a continuous animation cycle.

- ***Integration with Rendering Pipeline***
  The final animated positions were passed into the rendering pipeline, and lighting was recalculated per frame. This allowed us to display lit, animated wireframe scenes in real-time.

Demo:

In the final demo:

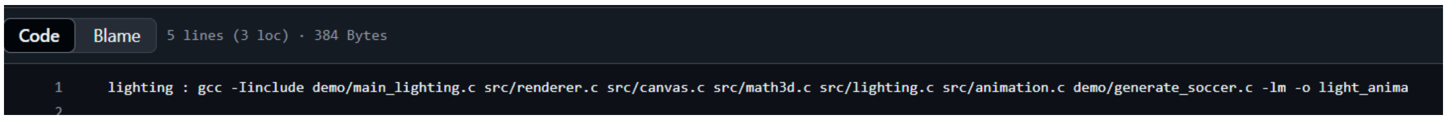
- Multiple wireframe objects (including the soccer ball) moved smoothly across the canvas along Bézier paths.
- Light sources were positioned to create shading variations based on object orientation.
- Animations were looped and synchronized, resulting in a polished and visually appealing presentation.

Challenges Faced:

1. Calculating accurate normal or directions for lines in wireframe models for lighting.
   so we implement two light after the two corner, and we implement the correctness shade the task required.
   so, then we got the correct out put frames and render it.
2. Finding correct coordinates of vertices and edges of the Truncated Icosahedron (Soccer Ball)
   Although the coordinates of vertices and edges were brought by some websites, these coordinates outputted faulty shapes. Therefore, the coordinates of the truncated icosahedron’s vertices and edges were calculated manually and added to the code implementation. Then, the expected shape of truncated icosahedron was outputted.

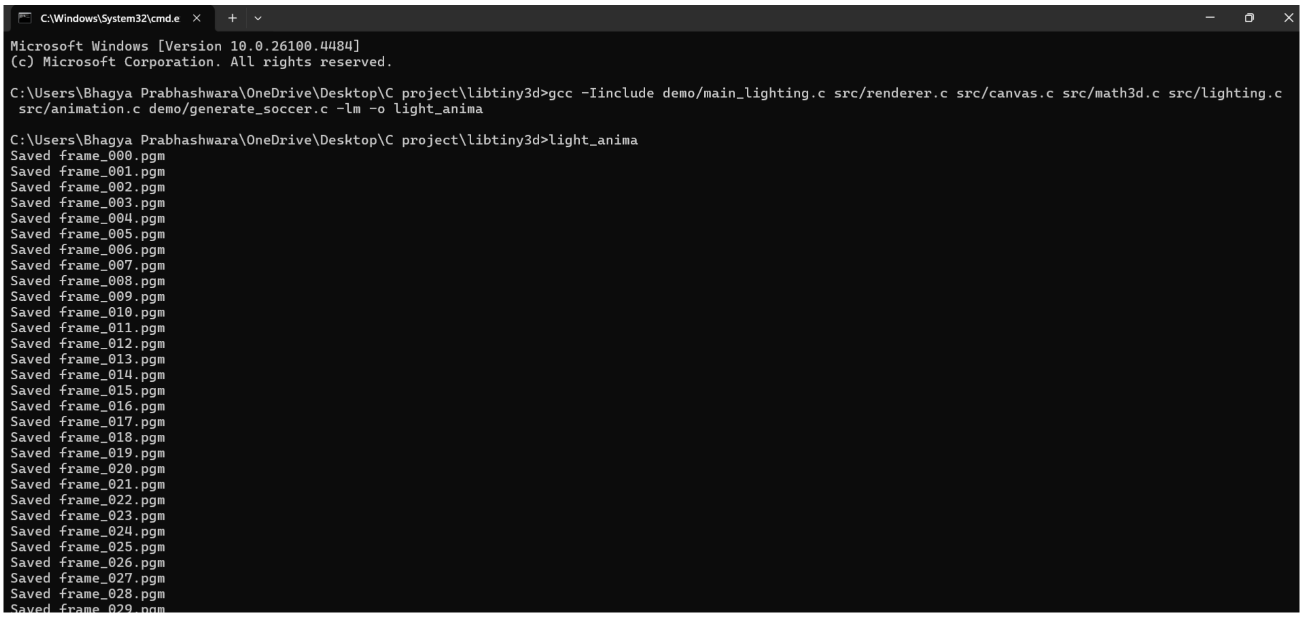
Task 4 run:

This is our cmd run command, and it gives 60 frames for the generate for rendering to achieve final output.

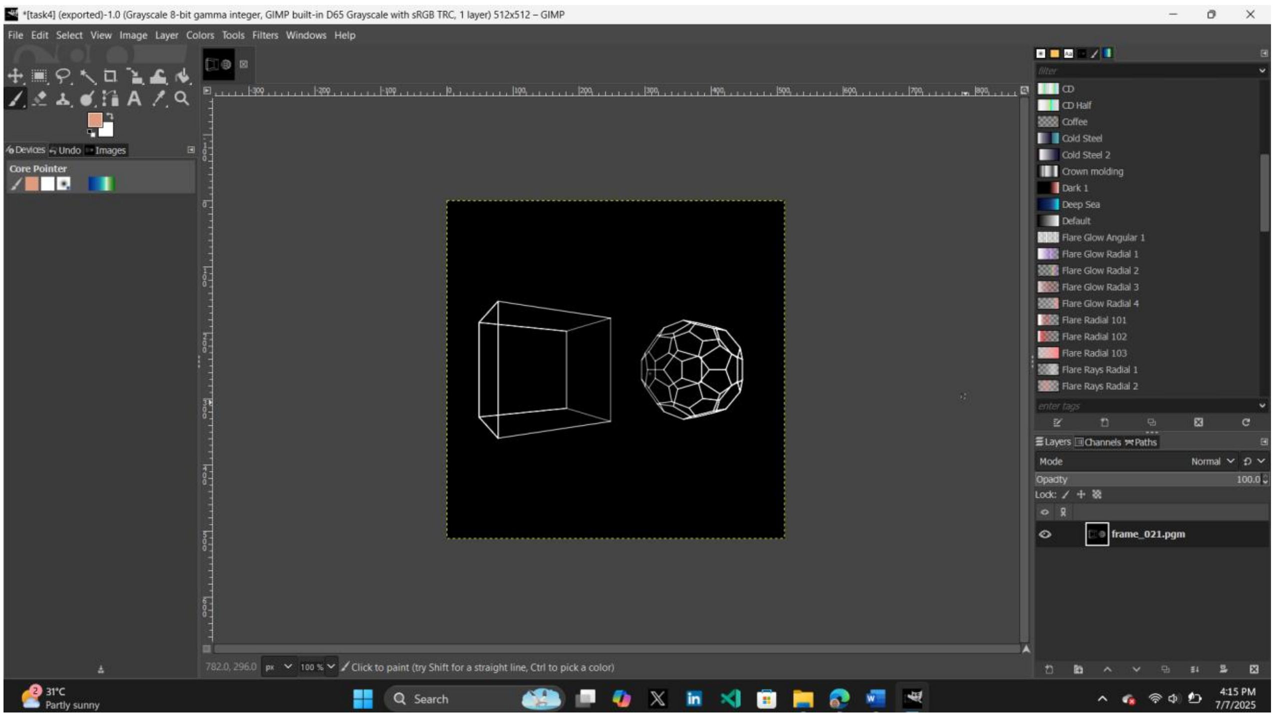


Code Blame 5 lines (3 loc) · 384 Bytes

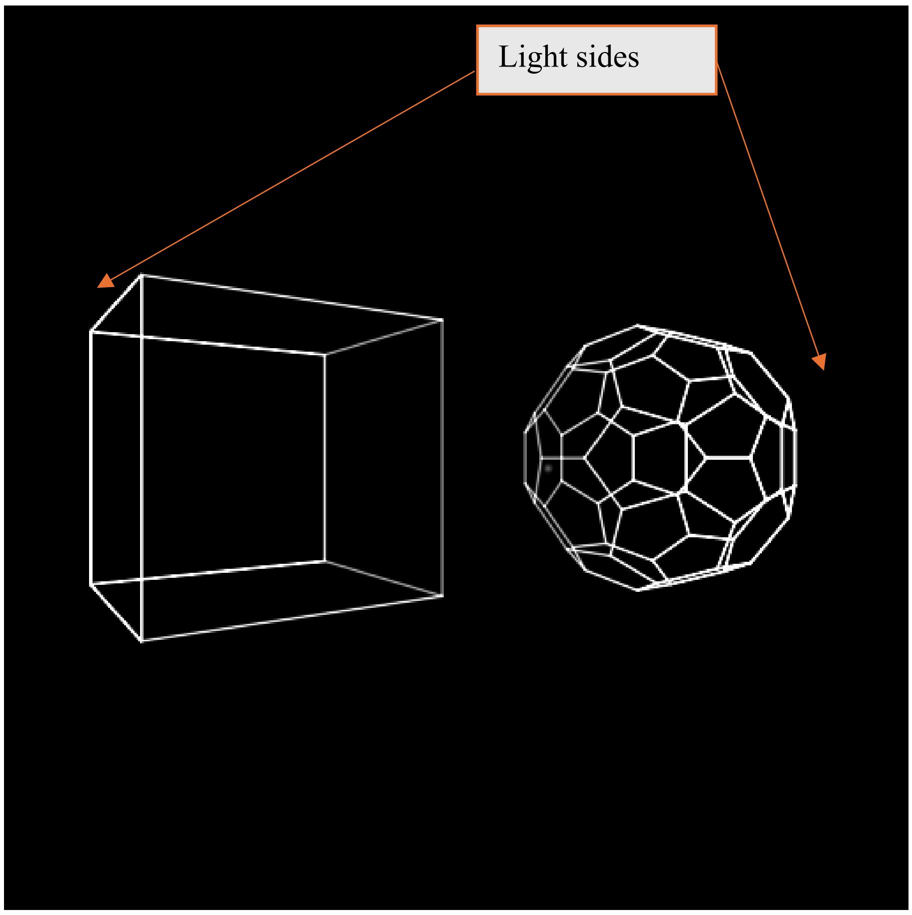
```
lighting : gcc -Iinclude demo/main_lighting.c src/renderer.c src/canvas.c src/math3d.c src/lighting.c src/animation.c demo/generate_soccer.c -lm -o light_anima
```



```
Microsoft Windows [Version 10.0.26100.4484]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>gcc -Iinclude demo/main_lighting.c src/renderer.c src/canvas.c src/math3d.c src/lighting.c
 src/animation.c demo/generate_soccer.c -lm -o light_anima

C:\Users\Bhagya Prabhashwara\OneDrive\Desktop\C project\libtiny3d>light_anima
Saved frame_000.pgm
Saved frame_001.pgm
Saved frame_002.pgm
Saved frame_003.pgm
Saved frame_004.pgm
Saved frame_005.pgm
Saved frame_006.pgm
Saved frame_007.pgm
Saved frame_008.pgm
Saved frame_009.pgm
Saved frame_010.pgm
Saved frame_011.pgm
Saved frame_012.pgm
Saved frame_013.pgm
Saved frame_014.pgm
Saved frame_015.pgm
Saved frame_016.pgm
Saved frame_017.pgm
Saved frame_018.pgm
Saved frame_019.pgm
Saved frame_020.pgm
Saved frame_021.pgm
Saved frame_022.pgm
Saved frame_023.pgm
Saved frame_024.pgm
Saved frame_025.pgm
Saved frame_026.pgm
Saved frame_027.pgm
Saved frame_028.pgm
Saved frame_029.pgm
```


*[task4] (exported)-1.0 (Grayscale 8-bit gamma integer, GIMP built-in D65 Grayscale with sRGB TRC, 1 layer) 512x512 – GIMP
File Edit Select View Image Layer Colors Tools Filters Windows Help
frame_021.pgm
Click to paint (try Shift for a straight line, Ctrl to pick a color)

This is our implemention : Rotating **lighting_animation**



Two corner side we are implementing lighting parts so its works, clearly.
And the middle of the canvas is growing dark side ,that how we implement the final task.

Challenges Faced:

3. Lighting implementation
   when we do this task first we face to problem is not clearly identify where is the lighting side and dark side, To reason that we first sit the lighting in front of the canvas, it not work so after we discuss and set it to the top of corner left and right then we could implement our task clearly. So, this is the way we faced that challenge.

DESIGN DECISIONS

To balance performance, clarity, and instructional value, we made several crucial design decisions throughout the creation of libtiny3d. Every choice was taken to adhere to the project's objective, which was to create a full 3D renderer entirely in C without the need of external graphics libraries, while maximizing code maintainability and technical accuracy.

- ***Programming Language: C***

  we chose pure C for this project. Because also it mention the CO1020 course module in additionally, it offers fine-grained control over memory and performance, which is crucial in rendering tasks.

- ***Matrix Layout: Column-Major Order:***

  We chose **column-major layout** for storing transformation matrices, aligning with industry standards like OpenGL. Although row-major is more intuitive for beginners, column-major simplifies matrix multiplication and chaining of transformations (e.g., scale → rotate → translate).

- ***Algorithms and techniques***:

  1. DDA for Line Drawing

  The Digital Differential Analyzer algorithm was selected for its simplicity and accuracy in drawing smooth lines between arbitrary points.

  2. Bilinear Filtering

  Applied in ***set_pixel_f()*** to support anti-aliasing by distributing intensity to neighboring pixels based on sub-pixel positions.

  3. SLERP and Bézier Curves

  These were used to ensure smooth transitions in orientation and movement, respectively, especially during animations.

- ***Z-Sorting over Full Z-Buffer***

  For simplicity and performance, we implemented **Z-sorting** (back-to-front rendering of edges) in wireframe rendering instead of a full Z-buffer. A full Z-buffer is more accurate for filled surfaces, but for wireframes, sorting lines based on depth was sufficient and less memory-intensive.

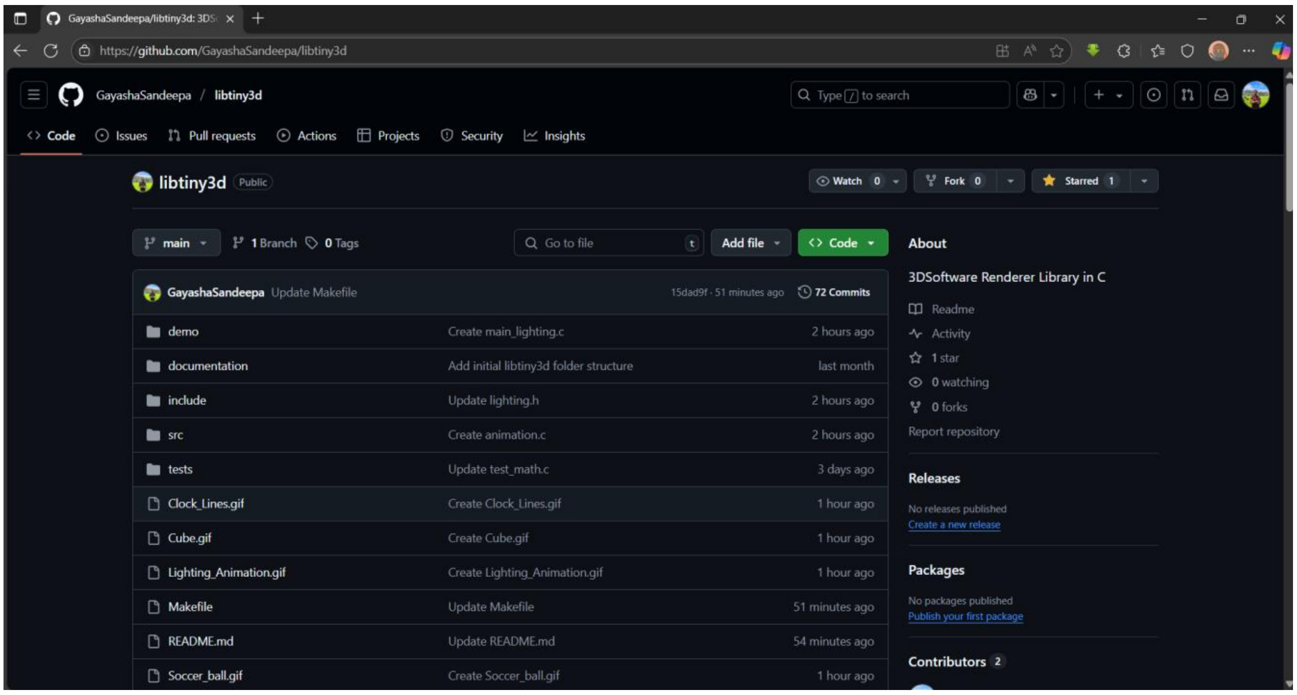
- ***Modularity and File Structure***

  The project was broken into clearly separated modules (**canvas.c**, **math3d.c**, **renderer.c**, **lighting.c**, etc.), each handling a well-defined responsibility. This modularity improved readability and debugging, Allowed parallel development across team members &Simplified testing of individual components

➢ *Project Structure:*

```
libtiny3d/
├─include/
│ ├─tiny3d.h
│ ├─canvas.h
│ ├─math3d.h
│ ├─renderer.h
│ └─lighting.h
├─src/
│ ├─canvas.c
│ ├─math3d.c
│ ├─renderer.c
│ └─lighting.c
├─tests/
│ ├─test_math.c
│ ├─test_pipeline.c
│ └─visual_tests/
├─demo/
│ └─main.c
├─build/
│ ├─libtiny3d.a
│ └─demo
├─documentation/
│ └─GroupXX_report.pdf
├─README.md
└─Makefile
```

This the given project structure in CO1020 module, also we following it add use additionally, GitHub for structuring this and we mange the project work given timeline.



GayashaSandeepa / libtiny3d

https://github.com/GayashaSandeepa/libtiny3d

libtiny3d Public

main · 1 Branch · 0 Tags

GayashaSandeepa Update Makefile · 15dad9f · 51 minutes ago · 72 Commits

demo	Create main_lighting.c	2 hours ago
documentation	Add initial libtiny3d folder structure	last month
include	Update lighting.h	2 hours ago
src	Create animation.c	2 hours ago
tests	Update test_math.c	3 days ago
Clock_Lines.gif	Create Clock_Lines.gif	1 hour ago
Cube.gif	Create Cube.gif	1 hour ago
Lighting_Animation.gif	Create Lighting_Animation.gif	1 hour ago
Makefile	Update Makefile	51 minutes ago
README.md	Update README.md	54 minutes ago
Soccer_ball.gif	Create Soccer_ball.gif	1 hour ago

About

3DSoftware Renderer Library in C

Readme · Activity · 1 star · 0 watching · 0 forks

Releases: No releases published

Packages: No packages published

Contributors 2

FIGURE: GitHub Repositories

INDIVIDUAL CONTRIBUTIONS & CHALLENGES FACED

Our team collaborated effectively throughout the four-week development of **libtiny3d**. Each member contributed to specific tasks based on their strengths and interest.
Our team is GroupNo:39
member 1: Gayasha Sandeepa. (E/22/353) GayashaSandeepa (Gayasha Sandeepa)
member 2: Bhagya Karunanayake. (E/22/184) zerokali20 (Bhagya Karunanayake)

Both of us build up the tasks and discussing some mathematical implementations otherwise we follow through basic knowledge and fundamental in first week and after we start the project tasks one by one, faced more challenges, because it new experience both of us, But we faced the battle successfully.

Basically we faced the implement task 3 because we spent more time to identify vertices and what needed edges to required and most time we use to identify rendering, otherwise we new to Gimp software and easily we got the idea behind thereover got the idea for rendering output frames and we implement the tasks successfully.
In addition to that we face to angle the lighting in canvas and to do for task 4.
we faced challenges but successfully we do the all task.

Go through this GitHub repo and check our implementations according libtiny3D project work.
Repo: GayashaSandeepa/libtiny3d: 3DSoftware Renderer Library in C

AI TOOL USAGE

We use the AI tools which call **ChatGPT** for some search cases.
basically we firstly it uses for drawing an outline our project within does for 4 weeks. so we get better timeline to successes. Falterer that we use find resources to go through. And use to learn how to handle GitHub with Gitbash.

we find some logical errors our coding parts using ChatGPT, not we implement the code or any functions using AI. But we use base ideas to find when task 3, handle vertices and edges. We found mathematical idea and use for it.

so that how we use AI tools for our libtiny3D project work.

CONCLUSION

The **libtiny3d** project was a challenging yet rewarding journey into the fundamentals of 3D computer graphics using pure C. Over the course of four weeks, we built a fully functional software-based 3D rendering engine from scratch, without relying on hardware acceleration or external graphics libraries.

Through this project, we gained a deep understanding of:

- Low-level pixel manipulation and anti-aliasing techniques
- The mathematical foundation of 3D transformations, including vectors, matrices, and interpolation
- The full 3D rendering pipeline from local object space to screen projection
- Lighting models (Lambertian) and smooth animation using Bézier curves
- Modular software architecture, testing strategies, and debugging large systems

By implementing every component manually including the math engine, wireframe renderer, and animation system we developed not only technical skills in systems programming, but also an appreciation for how modern 3D engines operate under the hood.

While there are still areas for future improvement, such as Z-buffering, texture mapping, and file-based geometry loading, the current version of libtiny3d successfully demonstrates the core principles of 3D rendering. Most importantly, it reflects our ability to design, implement, test, and integrate complex systems through teamwork and hands-on exploration.

This project has significantly enhanced our confidence in both graphics programming and C development and has laid a solid foundation for future work in game engines, simulation systems, or GPU-based rendering.

REFERENCES

1. Scratchapixel 2.0: https://scratchapixel.com
   (For understanding projection, lighting, and Bézier curves
   Your Starting Point!)
2. Bézier curve: Bézier curve - Wikipedia
3. OpenAI/ChatGPT: https://chatgpt.com/