LIBTINY3D

A Pure C Software 3D Graphics Engine

CO1020 — Computer Systems Programming

Department of Computer Engineering

University of Peradeniya

Submitted By:

Group 85

• M.D.S.Senanayake – E/22/366

• H.C.V.Perera – E/22/280

Content

1. Introduction

Embark with us on an adventure into the marvelous realm of low-level graphics with the libtiny3d project! This minimalist 3D graphics engine was meticulously hand-written in pure C, a testament to what can be achieved without the crutch of hardware-accelerated libraries like DirectX or OpenGL. Our foremost goal was to construct a whole software-based 3D rendering pipeline from the ground up using merely basic C functionality for file I/O, mathematics, and memory management.

Welcome to the heart of our "libtiny3d" project report! This project has been a real eye-opener into the abyss of low-level graphics programming. Building a 3D graphics engine from scratch, using only pure C and raw mathematical principles, has been an absolutely exhilarating experience. What excited our group the most was the ability to peel off the layers of abstraction provided by high-level libraries and truly understand how 3D worlds are realized, pixel by pixel. It has been a humbling experience in appreciating the subtle dance between math and computation that lies beneath all visual computing.

2. Project Overview

libtiny3d is a reaction from our company in accomplishing the task of designing a complete 3D graphics engine from scratch in four weeks. Our guiding philosophy to accomplish this work was to avoid usage of existing graphics libraries. Instead, we proceeded to design each component ourselves, from the underlying digital canvas to complex 3D math, rendering pipelines, lighting systems, and animation frameworks. This project was a deep exploration of the mathematical and computational foundation behind all 3D programs, games, and visualization tools, transforming abstract 3D geometric information into real 2D screen images.

3. Implementation Details

Task 1: Canvas & Line Drawing

Purpose

The objective of this exercise was to implement the ground-level piece of our rendering architecture-a quality, floating-point canvas that would be able to render anti-aliased lines. This module would serve as the base upon which the whole 3D rendering engine would be constructed.

Design

We defined a canvas_t structure to store our digital canvas. It contains attributes for the canvas height and width, and a 2D array of floating-point values between 0.0 (black) and 1.0 (white) representing the brightness of each pixel. This is not like traditional integer-based pixel systems in that it can provide smoother gradients and sub-pixel rendering.

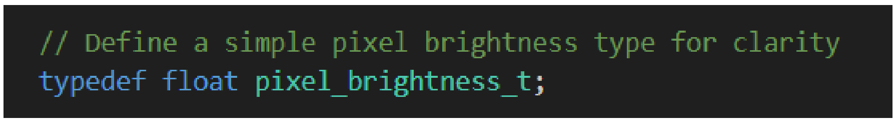
To enhance realism and eliminate jagged edges (aliasing), we used bilinear filtering in calculating pixel intensities. When drawing lines, we used the Digital Differential Analyzer (DDA) algorithm, which is straightforward but effective when rasterizing lines within a grid of pixels.

Implementations

File: canvas.h

canvas.h is employed to declare the interface for our 2D drawing system in our libtiny3d project. It sets up the underlying data structures and function prototypes needed for making, modifying, and exporting 2D images in terms of grayscale canvas outputs.

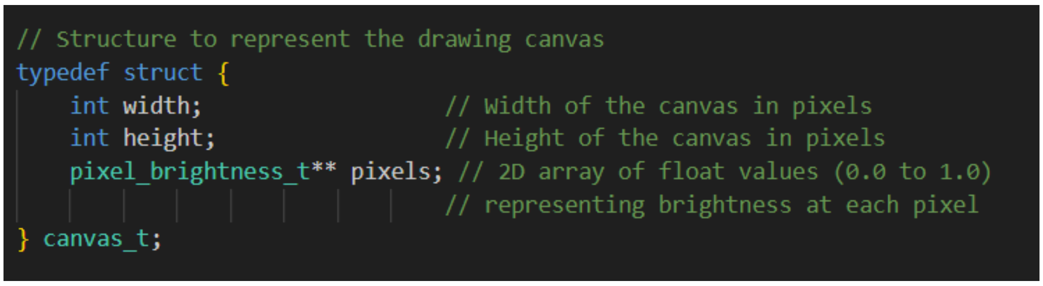
1. pixel_brightness_t



```c
// Define a simple pixel brightness type for clarity
typedef float pixel_brightness_t;
```

- Purpose: Represents the brightness of a single pixel on the canvas.
- Type: float between 0.0 (black) and 1.0 (white).
- Why float? Supports sub-pixel accuracy and smooth blending with bilinear filtering.

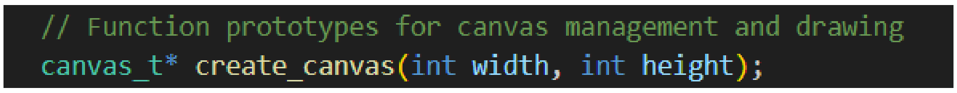
2. canvas_t structure



```c
// Structure to represent the drawing canvas
typedef struct {
    int width;                  // Width of the canvas in pixels
    int height;                 // Height of the canvas in pixels
    pixel_brightness_t** pixels; // 2D array of float values (0.0 to 1.0)
                                // representing brightness at each pixel
} canvas_t;
```

- width & height – Canvas size in pixels.
- pixels – Dynamically allocated 2D array of pixel brightness values.
- Each element: pixels[y][x] is a float (0.0 to 1.0) indicating how bright that pixel is.

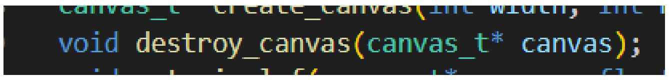
3. Function Prototypes
   These declare the drawing API functions, which are implemented in canvas.c:
   a. create_canvas(int width, int height)



```c
// Function prototypes for canvas management and drawing
canvas_t* create_canvas(int width, int height);
```

- Dynamically allocates memory and initializes an empty canvas of specified size.
- Sets all pixel values to 0.0 (black).

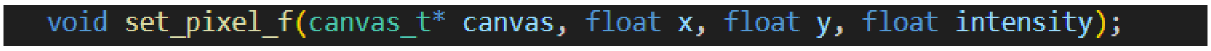
b. destroy_canvas(canvas_t* canvas)



```c
void destroy_canvas(canvas_t* canvas);
```

- Releases all memory associated with the canvas.
- Avoids memory leaks if the canvas is no longer needed.

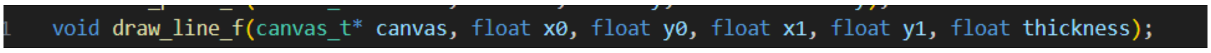
c. set_pixel_f(canvas_t* canvas, float x, float y, float intensity)



```c
void set_pixel_f(canvas_t* canvas, float x, float y, float intensity);
```

- Sets pixel intensity at a floating-point position (x, y) using bilinear filtering.
- Smooths brightness between adjacent pixels to avoid jagged lines.
- Supports sub-pixel rendering to obtain smoother graphics.

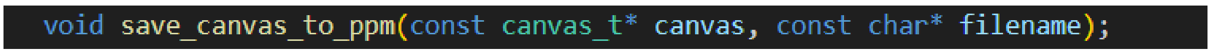
d. draw_line_f(canvas_t* canvas, float x0, float y0, float x1, float y1, float thickness)



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

- DRAWS a thick line from (x0, y0) to (x1, y1) using the DDA algorithm.
- Uses set_pixel_f() internally to create smooth line drawing.

e. save_canvas_to_ppm(const canvas_t* canvas, const char* filename)



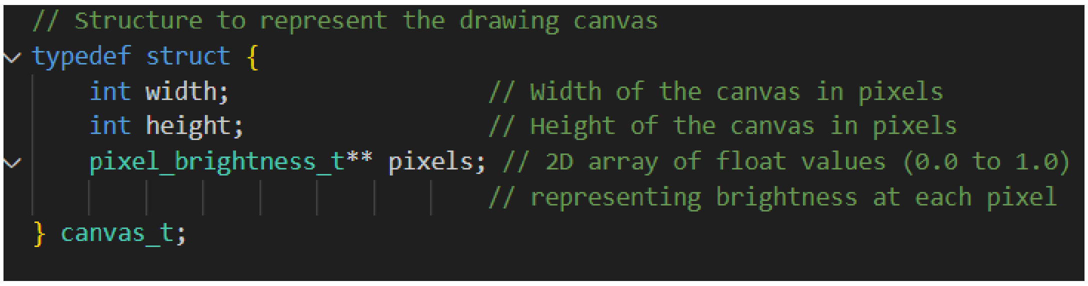
```c
void save_canvas_to_ppm(const canvas_t* canvas, const char* filename);
```

- Saves the canvas as a .ppm grayscale image file.
- Maps every float pixel value (0.0-1.0) to an 8-bit grayscale value (0-255).

File: canvas.c

canvas.c is utilized to initialize and manage a 2D drawing plane on which we can plot pixels and draw lines. This was a critical task to lay the foundation for our rendering system.

1. canvas_t Structure

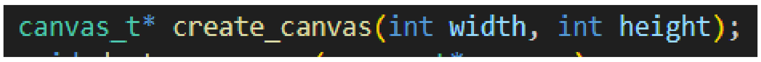


```c
// Structure to represent the drawing canvas
typedef struct {
    int width;                    // Width of the canvas in pixels
    int height;                   // Height of the canvas in pixels
    pixel_brightness_t** pixels;  // 2D array of float values (0.0 to 1.0)
                                  // representing brightness at each pixel
} canvas_t;
```

This structure holds:

- width and height: The size of the canvas.
- pixels: Dynamically allocated 2D array to store pixel brightness (float values between 0.0 and 1.0).

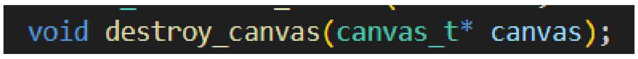
2. create_canvas(int width, int height)



```c
canvas_t* create_canvas(int width, int height);
```

- Changes the memory it allocates for the canvas and sets all pixels to 0.0 (black).
- Allocates row by row memory for 2D array of pixels.
- Why important: Dynamically initializes our canvas for drawing operations, provides resolution flexibility

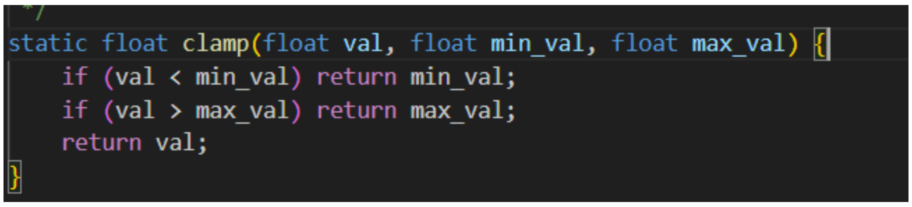
3. destroy_canvas(canvas_t* canvas)



```c
void destroy_canvas(canvas_t* canvas);
```

- Freewill allocates all dynamically allocated memory in reverse order.
- Prevents memory leaks after we're done with the canvas.

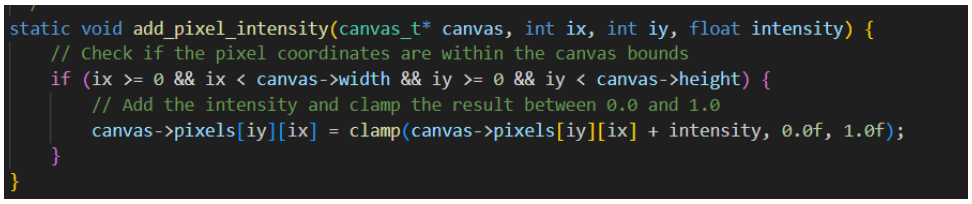
4. clamp(float val, float min_val, float max_val)



```c
static float clamp(float val, float min_val, float max_val) {
    if (val < min_val) return min_val;
    if (val > max_val) return max_val;
    return val;
}
```

- Maintains brightness values between 0.0 and 1.0.
- Prevents overbright or invalid values on updating pixels.

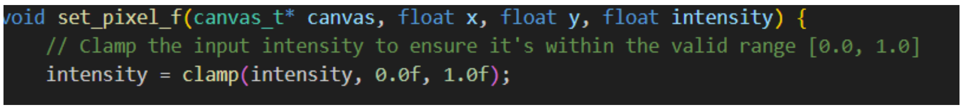
5. add_pixel_intensity(canvas, ix, iy, intensity)



```
static void add_pixel_intensity(canvas_t* canvas, int ix, int iy, float intensity) {
    // Check if the pixel coordinates are within the canvas bounds
    if (ix >= 0 && ix < canvas->width && iy >= 0 && iy < canvas->height) {
        // Add the intensity and clamp the result between 0.0 and 1.0
        canvas->pixels[iy][ix] = clamp(canvas->pixels[iy][ix] + intensity, 0.0f, 1.0f);
    }
}
```

- Adds intensity to an integer pixel and clamps the value.
- Does bounds checking to avoid writing off the canvas.
- Why important: Allows accumulation of brightness from multiple sources (e.g., bilinear filtering).

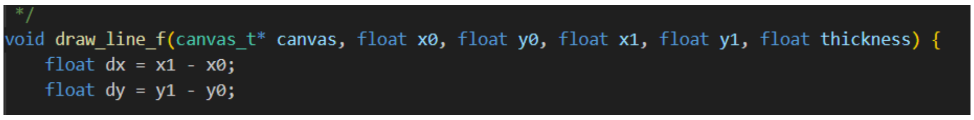
6. set_pixel_f(canvas, x, y, intensity)



```
void set_pixel_f(canvas_t* canvas, float x, float y, float intensity) {
    // Clamp the input intensity to ensure it's within the valid range [0.0, 1.0]
    intensity = clamp(intensity, 0.0f, 1.0f);
```

- Distributes intensity across the four nearest pixels using bilinear filtering.
- Weights based on the fractional part of (x, y).
  Key Formula Used:
  I(xf, yf) = I(x0, y0)*(1-Δx)*(1-Δy) + I(x1, y0)*Δx*(1-Δy) + ...
- Why important: Creates smoother graphics and sub-pixel accuracy while drawing points and lines.

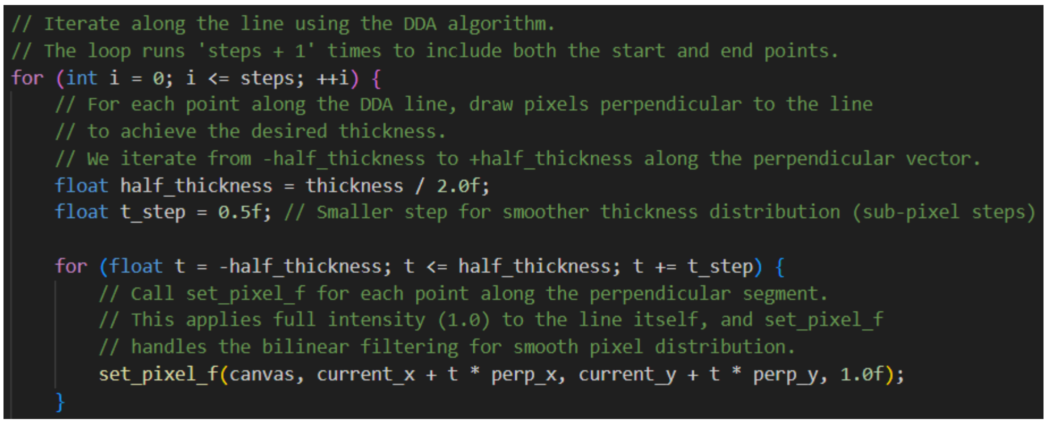
7. draw_line_f(canvas, x0, y0, x1, y1, thickness)



```
 */
void draw_line_f(canvas_t* canvas, float x0, float y0, float x1, float y1, float thickness) {
    float dx = x1 - x0;
    float dy = y1 - y0;
```

- Uses DDA algorithm.
- Draws additional pixels at each step along the direction perpendicular to the step in order to create an effect of thickness.

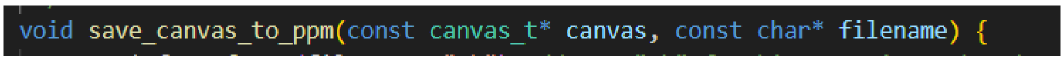
Loop concept:



```c
// Iterate along the line using the DDA algorithm.
// The loop runs 'steps + 1' times to include both the start and end points.
for (int i = 0; i <= steps; ++i) {
    // For each point along the DDA line, draw pixels perpendicular to the line
    // to achieve the desired thickness.
    // We iterate from -half_thickness to +half_thickness along the perpendicular vector.
    float half_thickness = thickness / 2.0f;
    float t_step = 0.5f; // Smaller step for smoother thickness distribution (sub-pixel steps)

    for (float t = -half_thickness; t <= half_thickness; t += t_step) {
        // Call set_pixel_f for each point along the perpendicular segment.
        // This applies full intensity (1.0) to the line itself, and set_pixel_f
        // handles the bilinear filtering for smooth pixel distribution.
        set_pixel_f(canvas, current_x + t * perp_x, current_y + t * perp_y, 1.0f);
    }
```

- Why important: Creates thick and precise lines with anti-aliasing effects.

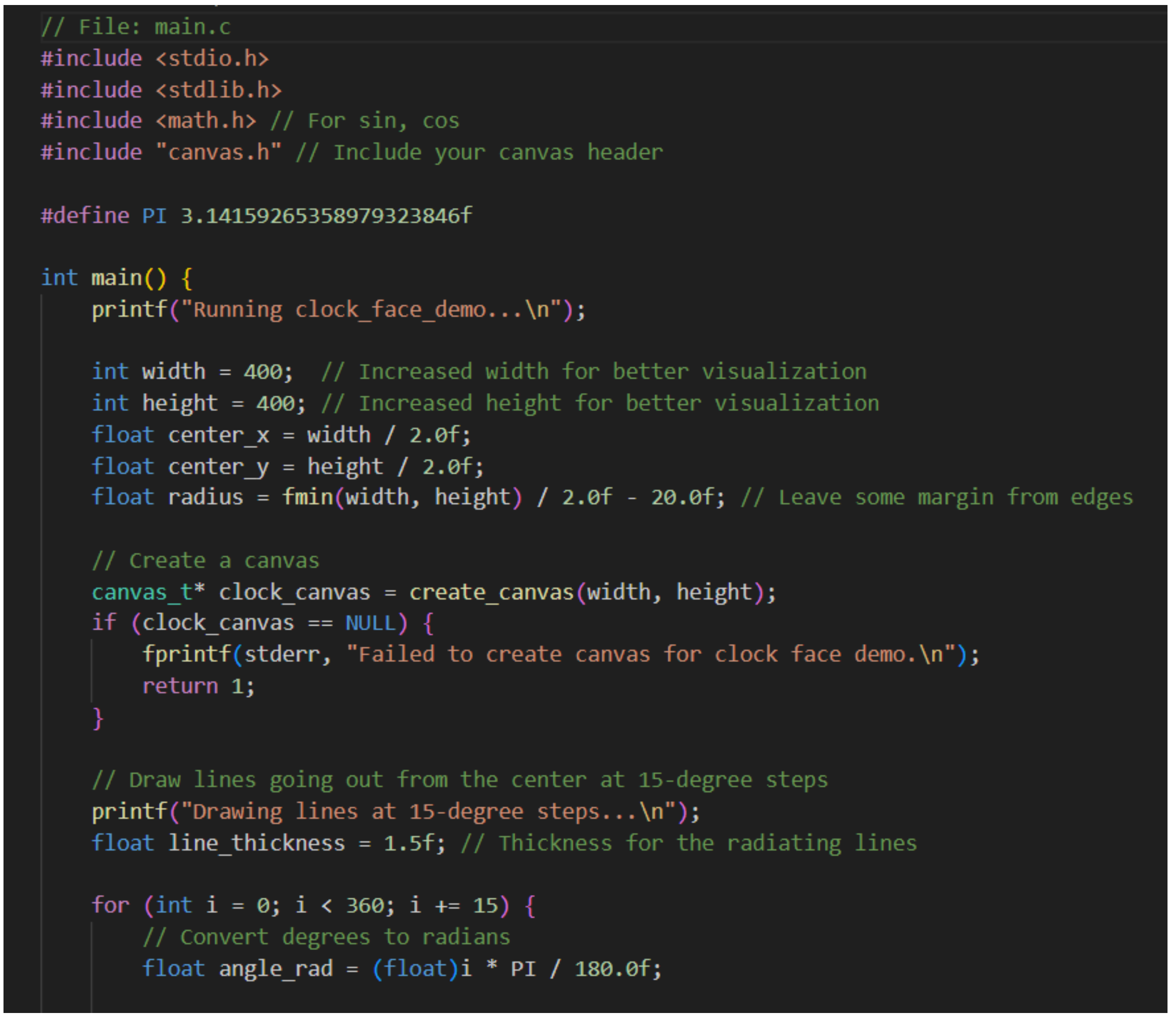
8. save_canvas_to_ppm(canvas, filename)



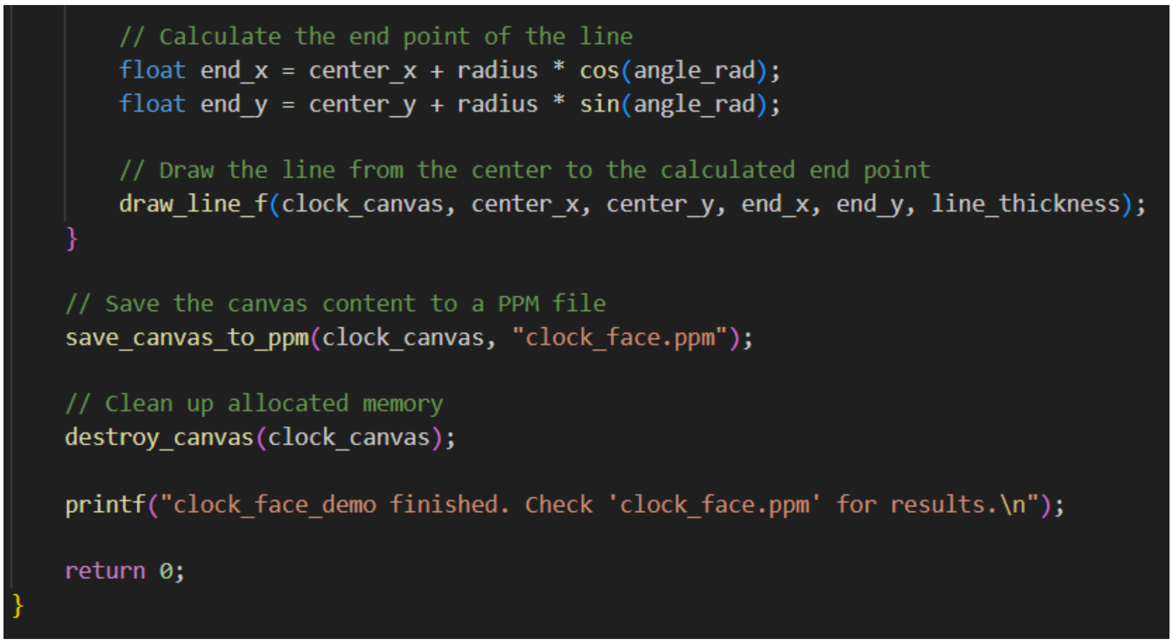
```c
void save_canvas_to_ppm(const canvas_t* canvas, const char* filename) {
```

- Saves canvas to .ppm file in ASCII format (P3).
- Scales brightness (0.0 to 1.0) to 8-bit grayscale (0–255).
- Significant since: Allows us to visually inspect for rendered frames as images.

File: main.c - Demo Program



```c
// File: main.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h> // For sin, cos
#include "canvas.h" // Include your canvas header

#define PI 3.14159265358979323846f

int main() {
    printf("Running clock_face_demo...\n");

    int width = 400;  // Increased width for better visualization
    int height = 400; // Increased height for better visualization
    float center_x = width / 2.0f;
    float center_y = height / 2.0f;
    float radius = fmin(width, height) / 2.0f - 20.0f; // Leave some margin from edges

    // Create a canvas
    canvas_t* clock_canvas = create_canvas(width, height);
    if (clock_canvas == NULL) {
        fprintf(stderr, "Failed to create canvas for clock face demo.\n");
        return 1;
    }

    // Draw lines going out from the center at 15-degree steps
    printf("Drawing lines at 15-degree steps...\n");
    float line_thickness = 1.5f; // Thickness for the radiating lines

    for (int i = 0; i < 360; i += 15) {
        // Convert degrees to radians
        float angle_rad = (float)i * PI / 180.0f;
```



```c
        // Calculate the end point of the line
        float end_x = center_x + radius * cos(angle_rad);
        float end_y = center_y + radius * sin(angle_rad);

        // Draw the line from the center to the calculated end point
        draw_line_f(clock_canvas, center_x, center_y, end_x, end_y, line_thickness);
    }

    // Save the canvas content to a PPM file
    save_canvas_to_ppm(clock_canvas, "clock_face.ppm");

    // Clean up allocated memory
    destroy_canvas(clock_canvas);

    printf("clock_face_demo finished. Check 'clock_face.ppm' for results.\n");

    return 0;
}
```

To show the capabilities of our line drawing implementation, we created a clock-face demo:

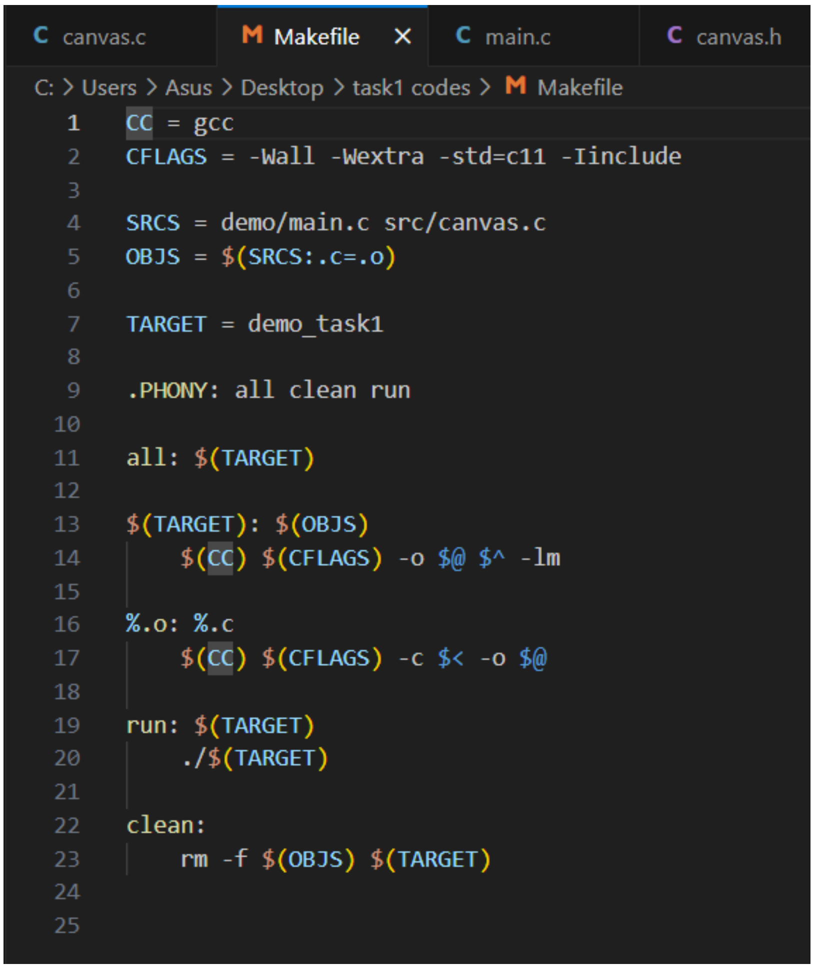
- Canvas size: 400 × 400
- Center: (200, 200)
- Radius: min(width, height)/2 – 20
- Line thickness: 1.5
- Lines drawn every 15° from the center outwards in a 24-line radial pattern.

clock_canvas was then saved as clock_face.ppm.

Compilation & Execution

A simple Makefile was created to compile and run the demo:

Makefile

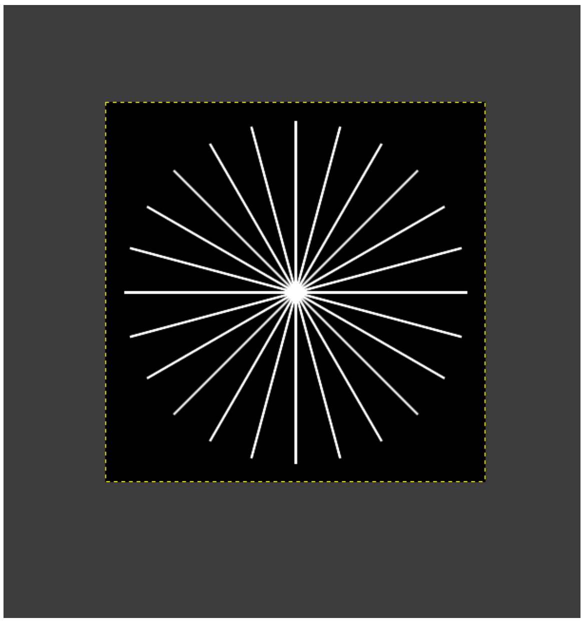


```
CC = gcc
CFLAGS = -Wall -Wextra -std=c11 -Iinclude

SRCS = demo/main.c src/canvas.c
OBJS = $(SRCS:.c=.o)

TARGET = demo_task1

.PHONY: all clean run

all: $(TARGET)

$(TARGET): $(OBJS)
	$(CC) $(CFLAGS) -o $@ $^ -lm

%.o: %.c
	$(CC) $(CFLAGS) -c $< -o $@

run: $(TARGET)
	./$(TARGET)

clean:
	rm -f $(OBJS) $(TARGET)
```

```
make            // Compiles demo/main.c and src/canvas.c
make run        // Executes the demo
make clean      // Cleans up object files and binary
```

Output

This output of this task was the image file:



clock_face.ppm

When viewed in a PPM-aware image viewer, the file displays:

- A smooth, evenly spaced radial line pattern
- Crisp anti-aliased lines
- Sub-pixel accurate rendering
- A visual appearance near that of an analog clock face

This verifies the validity of both our set_pixel_f() and draw_line_f() implementations and illustrates that our canvas system can be utilized to create clean and versatile line-based rendering.

Task 2: 3D Math Foundation

Purpose

A robust 3D graphics engine requires a solid mathematical background to properly manipulate and translate objects in 3D space. In this exercise, we implemented a customized math engine for 3D mathematics that does vector math and 4×4 matrix transformations. This mathematics library is required to perform translations, rotations, scalings, and projections—operations that are crucial in any 3D rendering pipeline.

Design

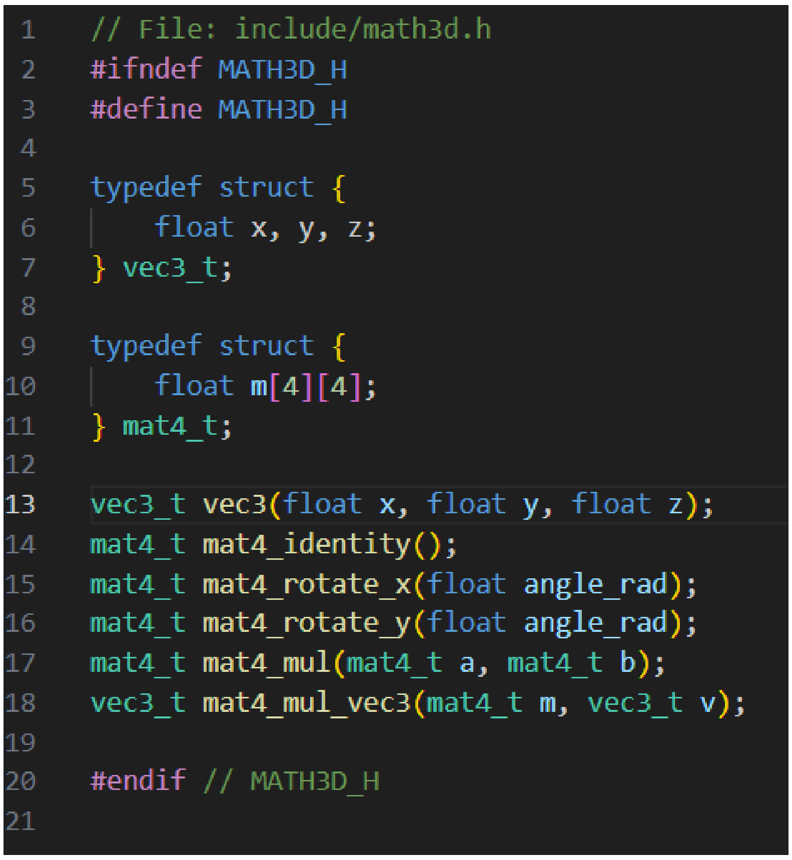
Our mathematics module includes two basic structures: vec3_t and mat4_t. The vec3_t structure includes operations for vectors within 3D space, while mat4_t operates on 4×4 transformation matrices in column-major order to maintain 3D graphics convention compatibility.

We provide functions for:

- Creation of vectors and matrices
- Vector normalizing and interpolating
- Matrix creation and combination (rotation, identity, etc.)
- Multiplying vectors by transformation matrices

Implementation

math3d.h file: Structure Declarations & Prototypes



```c
// File: include/math3d.h
#ifndef MATH3D_H
#define MATH3D_H

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

typedef struct {
    float m[4][4];
} mat4_t;

vec3_t vec3(float x, float y, float z);
mat4_t mat4_identity();
mat4_t mat4_rotate_x(float angle_rad);
mat4_t mat4_rotate_y(float angle_rad);
mat4_t mat4_mul(mat4_t a, mat4_t b);
vec3_t mat4_mul_vec3(mat4_t m, vec3_t v);

#endif // MATH3D_H
```

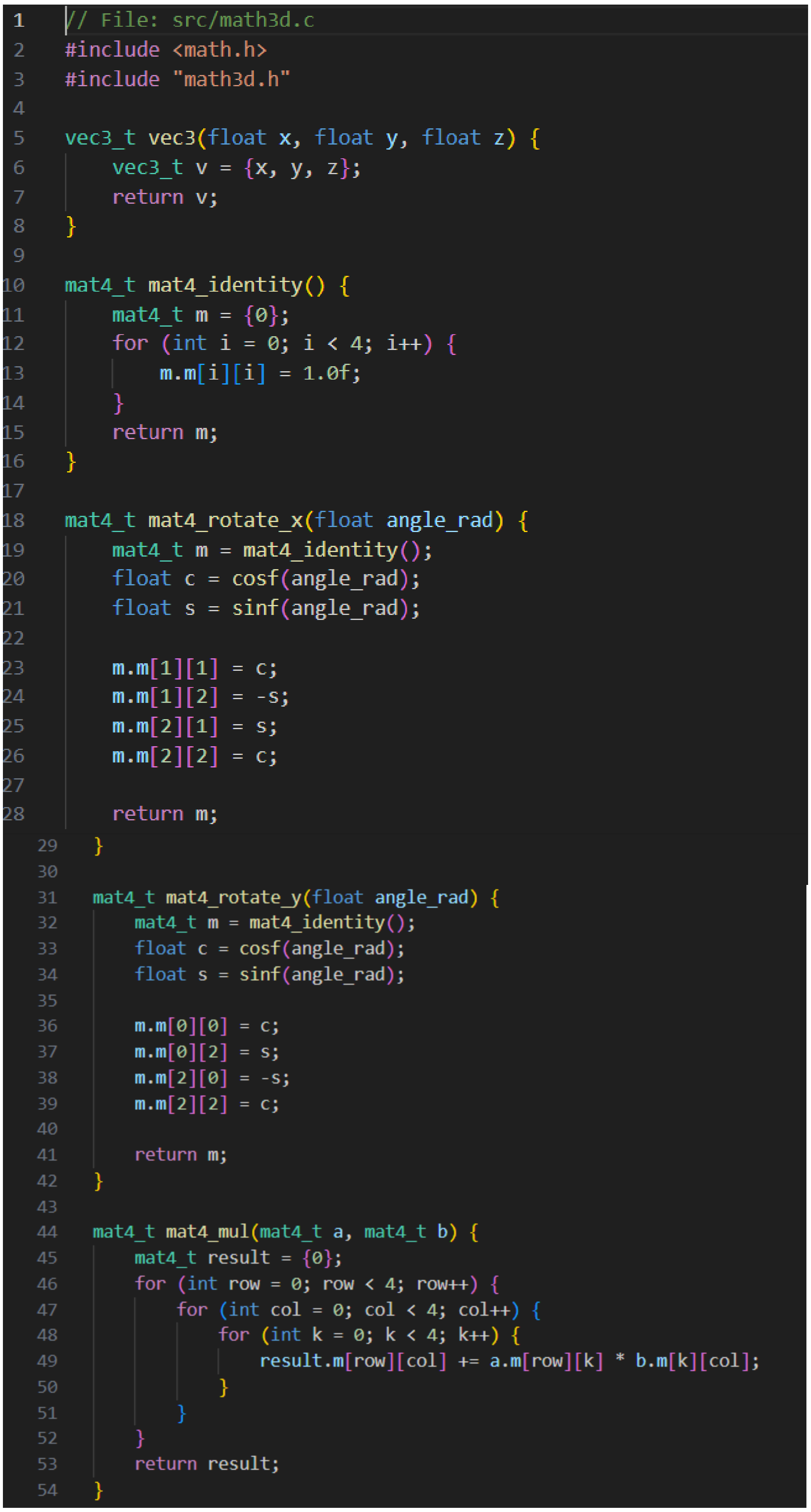
We defined:

- vec3_t: to store (x, y, z) coordinates
- mat4_t: a 4x4 float matrix for transforms

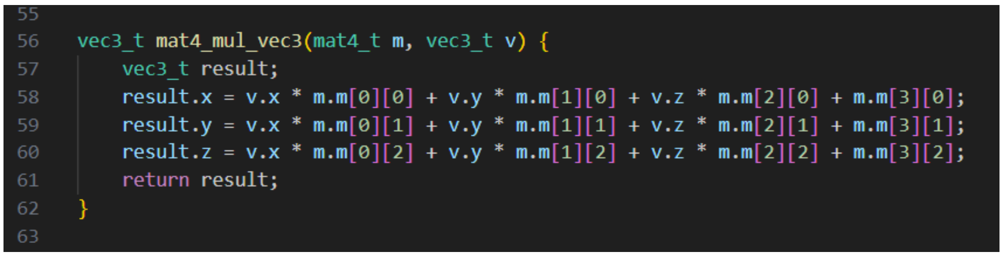
Function prototypes:

- vec3() – builds a 3D vector
- mat4_identity() – generates an identity matrix
- mat4_rotate_x() / mat4_rotate_y() – generates rotation matrices
- mat4_mul() – multiply two 4×4 matrices
- mat4_mul_vec3() – multiply a transformation matrix with a vector

math3d.c file: Core Math Logic



```c
// File: src/math3d.c
#include <math.h>
#include "math3d.h"

vec3_t vec3(float x, float y, float z) {
    vec3_t v = {x, y, z};
    return v;
}

mat4_t mat4_identity() {
    mat4_t m = {0};
    for (int i = 0; i < 4; i++) {
        m.m[i][i] = 1.0f;
    }
    return m;
}

mat4_t mat4_rotate_x(float angle_rad) {
    mat4_t m = mat4_identity();
    float c = cosf(angle_rad);
    float s = sinf(angle_rad);

    m.m[1][1] = c;
    m.m[1][2] = -s;
    m.m[2][1] = s;
    m.m[2][2] = c;

    return m;
}

mat4_t mat4_rotate_y(float angle_rad) {
    mat4_t m = mat4_identity();
    float c = cosf(angle_rad);
    float s = sinf(angle_rad);

    m.m[0][0] = c;
    m.m[0][2] = s;
    m.m[2][0] = -s;
    m.m[2][2] = c;

    return m;
}

mat4_t mat4_mul(mat4_t a, mat4_t b) {
    mat4_t result = {0};
    for (int row = 0; row < 4; row++) {
        for (int col = 0; col < 4; col++) {
            for (int k = 0; k < 4; k++) {
                result.m[row][col] += a.m[row][k] * b.m[k][col];
            }
        }
    }
    return result;
}
```



```c
vec3_t mat4_mul_vec3(mat4_t m, vec3_t v) {
    vec3_t result;
    result.x = v.x * m.m[0][0] + v.y * m.m[1][0] + v.z * m.m[2][0] + m.m[3][0];
    result.y = v.x * m.m[0][1] + v.y * m.m[1][1] + v.z * m.m[2][1] + m.m[3][1];
    result.z = v.x * m.m[0][2] + v.y * m.m[1][2] + v.z * m.m[2][2] + m.m[3][2];
    return result;
}
```

In source code, we place the actual logic there:

- vec3() creates a new 3D vector
- mat4_identity() returns an identity matrix with 1s on diagonal
- mat4_rotate_x() and mat4_rotate_y() construct rotation matrices for X and Y axes based on sine and cosine
- mat4_mul() does matrix multiplication using triple for-loops
- mat4_mul_vec3() uses the transformation matrix to apply transformation on a vector

Demo: Cube Rotation

cube_demo.c

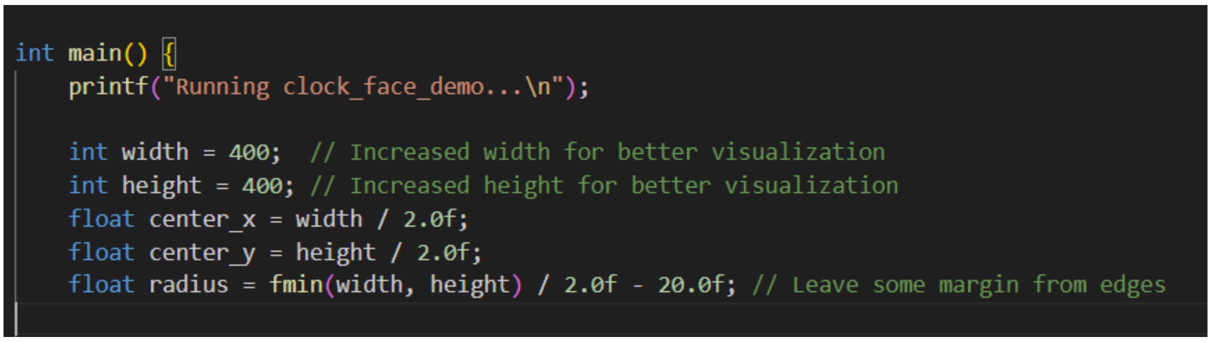
This demo aims to test our line-drawing and canvas-rendering systems by making a clock-like radial pattern.

```c
// File: main.c
#include <stdio.h>
#include <stdlib.h>
#include <math.h> // For sin, cos
#include "canvas.h" // Include your canvas header
```

- Shared headers provide common functionality.
- canvas.h is our own API to draw on a 2D float-based pixel canvas.

```c
#define PI 3.14159265358979323846f
```

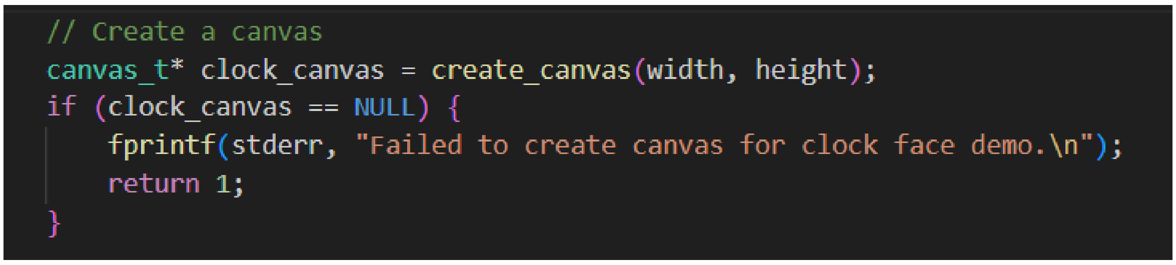
- Defines the constant π for conversion of angles from degrees to radians.



```c
int main() {
    printf("Running clock_face_demo...\n");

    int width = 400;  // Increased width for better visualization
    int height = 400; // Increased height for better visualization
    float center_x = width / 2.0f;
    float center_y = height / 2.0f;
    float radius = fmin(width, height) / 2.0f - 20.0f; // Leave some margin from edges
```

- We create an initial square canvas (400×400).
- The origin of the canvas is at its center when drawing lines.
- A margin is also achieved by reducing the radius to prevent overflow of lines.

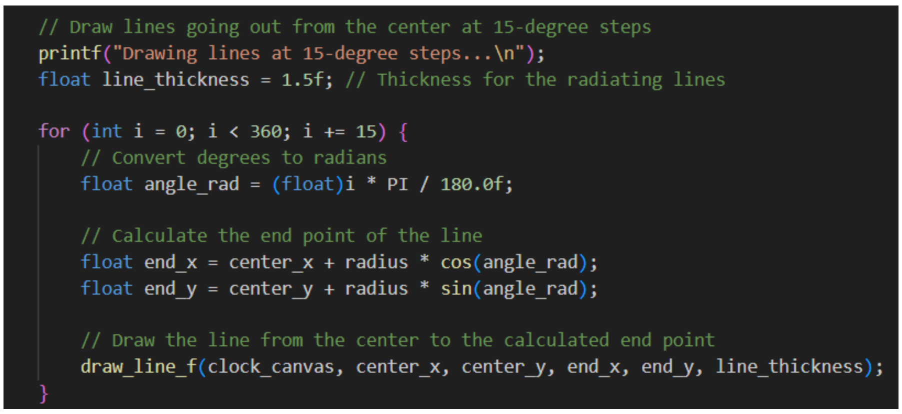
Canvas Creation



```c
    // Create a canvas
    canvas_t* clock_canvas = create_canvas(width, height);
    if (clock_canvas == NULL) {
        fprintf(stderr, "Failed to create canvas for clock face demo.\n");
        return 1;
    }
```

- The create_canvas() function initializes and allocates the pixel buffer.
- All pixel values are initialized to 0.0f (black).

Drawing Radial Lines (like a clock)



```c
    // Draw lines going out from the center at 15-degree steps
    printf("Drawing lines at 15-degree steps...\n");
    float line_thickness = 1.5f; // Thickness for the radiating lines

    for (int i = 0; i < 360; i += 15) {
        // Convert degrees to radians
        float angle_rad = (float)i * PI / 180.0f;

        // Calculate the end point of the line
        float end_x = center_x + radius * cos(angle_rad);
        float end_y = center_y + radius * sin(angle_rad);

        // Draw the line from the center to the calculated end point
        draw_line_f(clock_canvas, center_x, center_y, end_x, end_y, line_thickness);
    }
```

- We loop from 0° to 345° in steps of 15° (like clock ticks).
- Each angle is converted to radians.
- The (end_x, end_y) coordinates are calculated using trigonometry.
- The draw_line_f() function draws smooth anti-aliased lines with bilinear filtering and the DDA algorithm.

Saving Output as Image

```
// Save the canvas content to a PPM file
save_canvas_to_ppm(clock_canvas, "clock_face.ppm");
```

- Canvas content is output to a .ppm image file in grayscale mode.
- Pixel intensities are scaled into the 0–255 range for display.

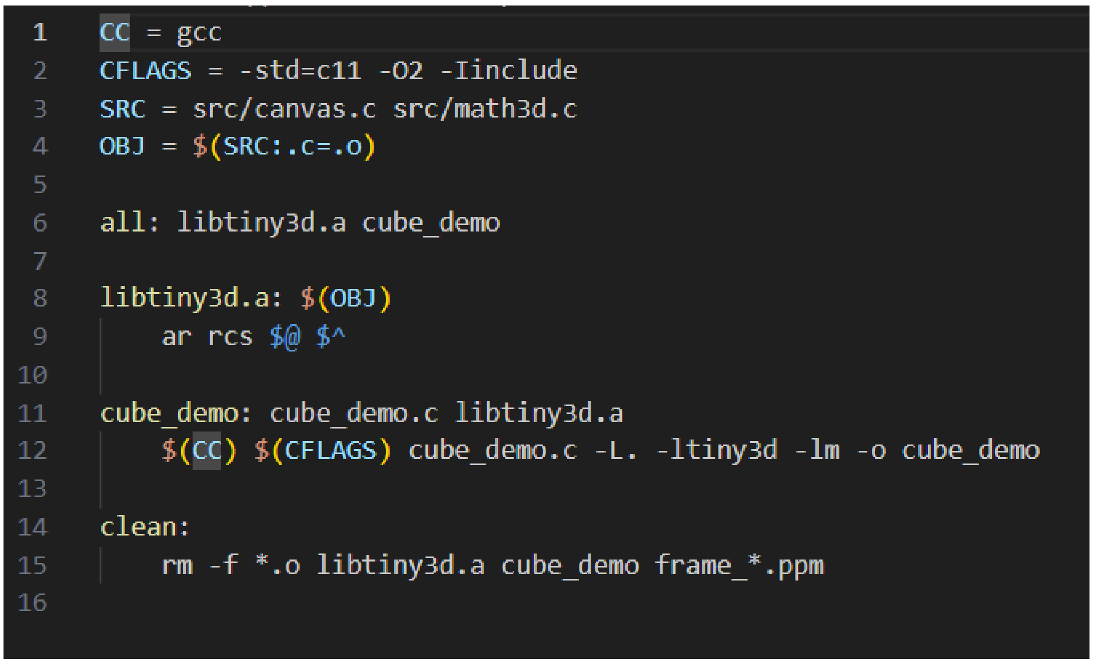
Cleanup

```
// Clean up allocated memory
destroy_canvas(clock_canvas);
```

- Frees all dynamically allocated memory related to the canvas.

Compilation & Execution

Makefile



```
CC = gcc
CFLAGS = -std=c11 -O2 -Iinclude
SRC = src/canvas.c src/math3d.c
OBJ = $(SRC:.c=.o)

all: libtiny3d.a cube_demo

libtiny3d.a: $(OBJ)
	ar rcs $@ $^

cube_demo: cube_demo.c libtiny3d.a
	$(CC) $(CFLAGS) cube_demo.c -L. -ltiny3d -lm -o cube_demo

clean:
	rm -f *.o libtiny3d.a cube_demo frame_*.ppm
```

We created a Makefile to build the math library (libtiny3d.a) and the demo application. The key targets are:

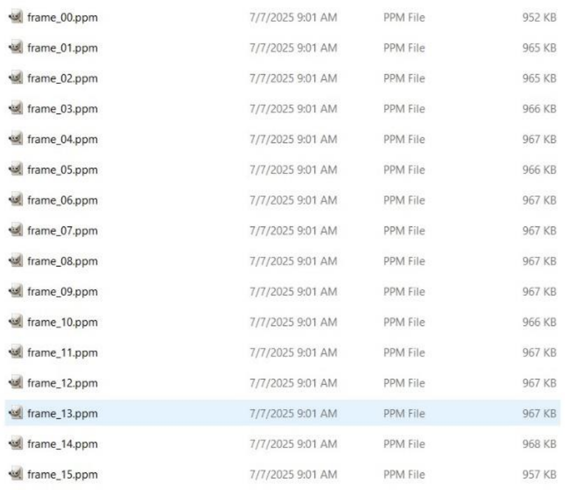
- all – builds static library and cube demo
- cube_demo – links project and builds executable
- clean – deletes build products and ppm images

Output

In Task 3.2, we set up a simple demo to visually confirm our 3D math engine by rendering a rotating cube via matrix transformations and projection. We hard-coded the cube with 8 vertices and 12 edges and rotated it smoothly over time using our own mat4_rotate_x() and mat4_rotate_y() functions.

Each frame involved multiplying a transformation matrix, projecting the 3D points to 2D screen space, and drawing lines between them using our canvas module's draw_line_f().

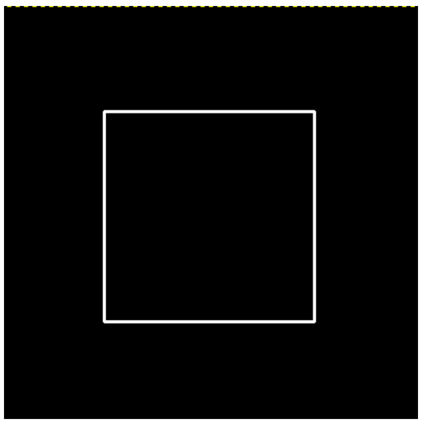
- Frame resolution: 400 × 400 pixels
- Number of frames generated: 60
- Output filenames:



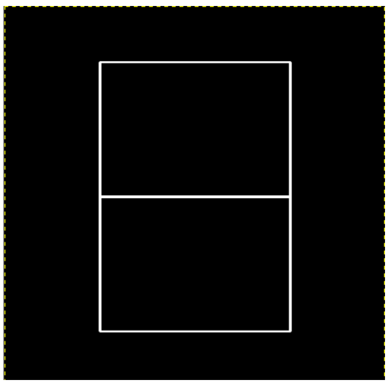
frame_00.ppm	7/7/2025 9:01 AM	PPM File	952 KB
frame_01.ppm	7/7/2025 9:01 AM	PPM File	965 KB
frame_02.ppm	7/7/2025 9:01 AM	PPM File	965 KB
frame_03.ppm	7/7/2025 9:01 AM	PPM File	966 KB
frame_04.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_05.ppm	7/7/2025 9:01 AM	PPM File	966 KB
frame_06.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_07.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_08.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_09.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_10.ppm	7/7/2025 9:01 AM	PPM File	966 KB
frame_11.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_12.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_13.ppm	7/7/2025 9:01 AM	PPM File	967 KB
frame_14.ppm	7/7/2025 9:01 AM	PPM File	968 KB
frame_15.ppm	7/7/2025 9:01 AM	PPM File	957 KB

Sample Output Frames

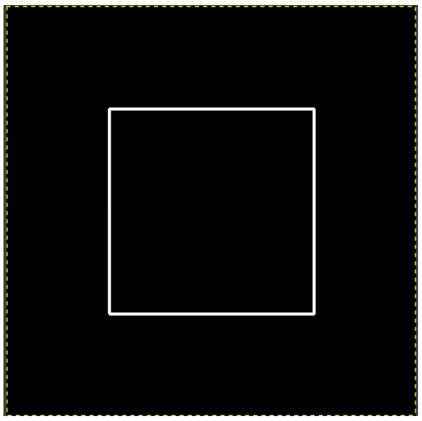
- Frame 0



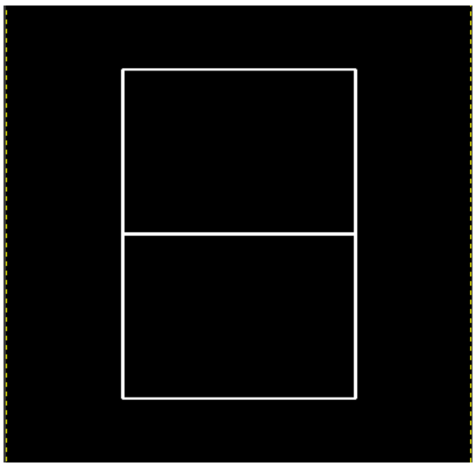
- Frame 15



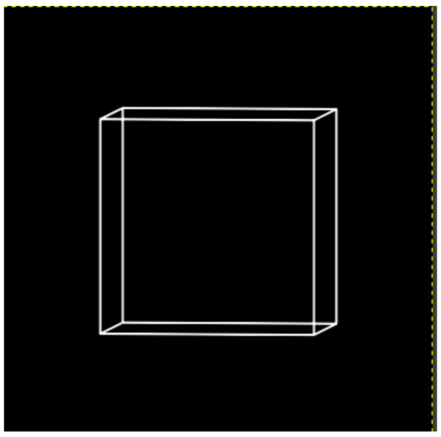
- Frame 30



- Frame 45



- Frame 59



This output validates the correctness of our matrix operations, vector transformations, and projection logic. Each rendered frame demonstrates:

- Smooth and continuous cube rotation via matrix multiplication.
- Stable 2D projection from 3D space.
- Sub-pixel accurate rendering of clean lines using the draw_line_f() function via wireframe rendering.
- Complete 60-frame loop that renders a smoothly rotating 3D cube.
- These results confirm the stability and accuracy of our 3D math foundation, providing a solid foundation for all future 3D rendering features in the libtiny3d pipeline.

Task 3: 3D Rendering Pipeline

Purpose

This task was to implement a complete 3D rendering pipeline that would be able to render wireframe models (like a soccer ball) onto a 2D canvas. We implemented vertex transformation, projection, circular viewport clipping, and line-based rendering, and then added smooth cubic Bezier animation to produce rotating and animated 3D object.

Desing

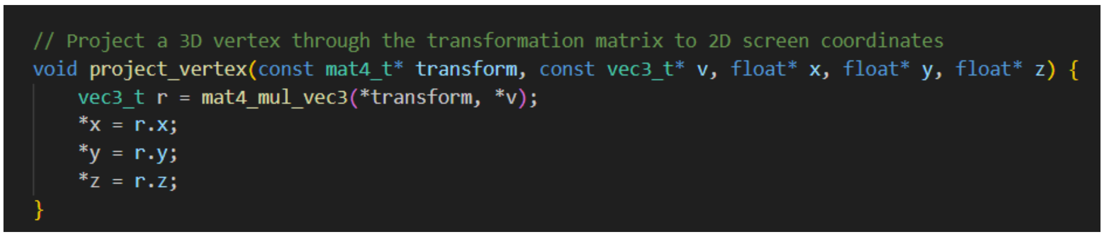
Our pipeline consists of the following transformation steps:

- Local to World Transformation - Does object rotation via matrix multiplication.
- Projection to Screen - Converts 3D space to 2D canvas coordinates.
- Clipping - Rejects objects outside a circular viewport.
- Wireframe Rendering - Renders visible edges with anti-aliased lines.
- Bezier Animation - Animates object position smoothly with cubic Bezier curves.

Implementation

<u>renderer.c file:</u>

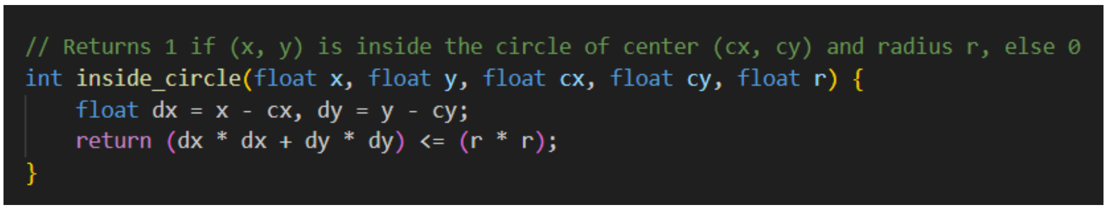
1. Vertex Projection



```c
// Project a 3D vertex through the transformation matrix to 2D screen coordinates
void project_vertex(const mat4_t* transform, const vec3_t* v, float* x, float* y, float* z) {
    vec3_t r = mat4_mul_vec3(*transform, *v);
    *x = r.x;
    *y = r.y;
    *z = r.z;
}
```

This routine projects a 3D vertex via the 4×4 transformation matrix and stores the outcome as 2D coordinates (x, y) along with a depth z. It's the heart of world-to-screen conversion.

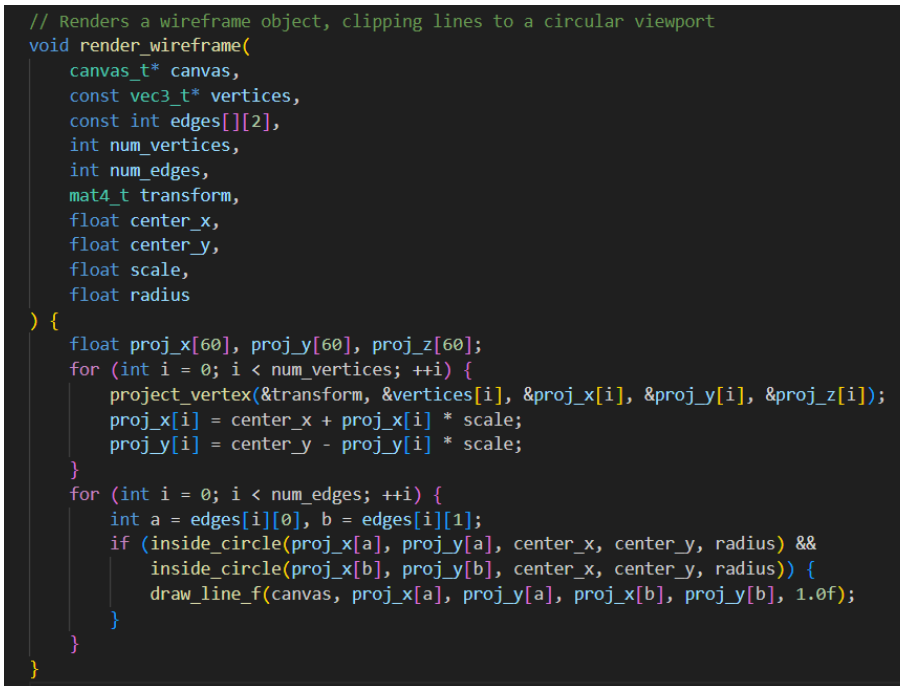
2. Viewport Clipping



```c
// Returns 1 if (x, y) is inside the circle of center (cx, cy) and radius r, else 0
int inside_circle(float x, float y, float cx, float cy, float r) {
    float dx = x - cx, dy = y - cy;
    return (dx * dx + dy * dy) <= (r * r);
}
```

It tests if a point is within the circular viewport defined by center (cx, cy) and radius r. This limits the rendering to a circular region.

3. Wireframe Rendering

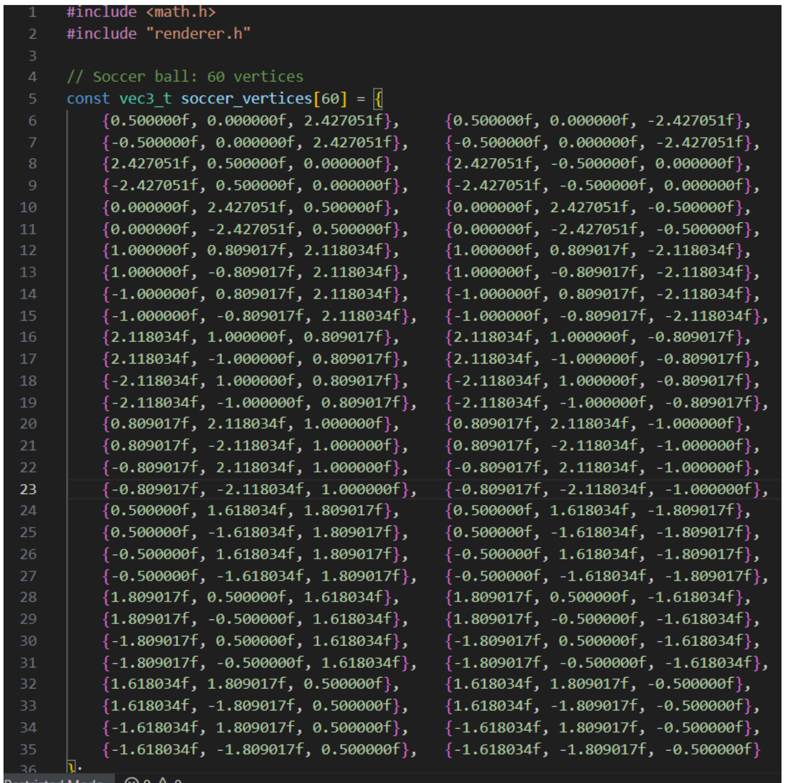


```c
// Renders a wireframe object, clipping lines to a circular viewport
void render_wireframe(
    canvas_t* canvas,
    const vec3_t* vertices,
    const int edges[][2],
    int num_vertices,
    int num_edges,
    mat4_t transform,
    float center_x,
    float center_y,
    float scale,
    float radius
) {
    float proj_x[60], proj_y[60], proj_z[60];
    for (int i = 0; i < num_vertices; ++i) {
        project_vertex(&transform, &vertices[i], &proj_x[i], &proj_y[i], &proj_z[i]);
        proj_x[i] = center_x + proj_x[i] * scale;
        proj_y[i] = center_y - proj_y[i] * scale;
    }
    for (int i = 0; i < num_edges; ++i) {
        int a = edges[i][0], b = edges[i][1];
        if (inside_circle(proj_x[a], proj_y[a], center_x, center_y, radius) &&
            inside_circle(proj_x[b], proj_y[b], center_x, center_y, radius)) {
            draw_line_f(canvas, proj_x[a], proj_y[a], proj_x[b], proj_y[b], 1.0f);
        }
    }
}
```

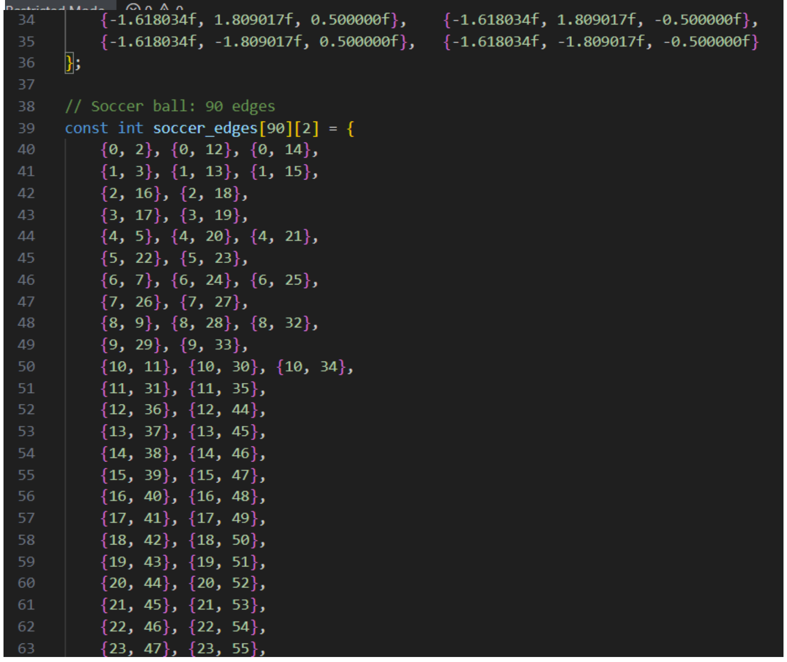
This routine:

- Applies all 3D vertices to the combined rotation matrix.
- Projects onto 2D space.
- Clips edges outside of circular viewport.
- Draws lines at sub-pixel resolution using draw_line_f().

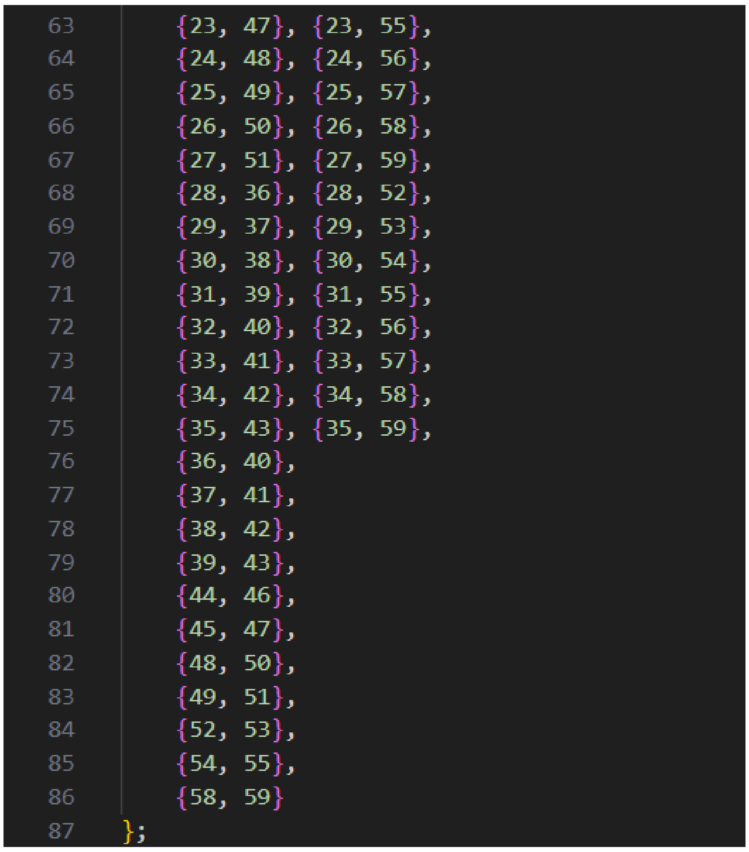
4. Soccer Ball Model



```c
#include <math.h>
#include "renderer.h"

// Soccer ball: 60 vertices
const vec3_t soccer_vertices[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},
    {0.000000f, 2.427051f, 0.500000f},     {0.000000f, 2.427051f, -0.500000f},
    {0.000000f, -2.427051f, 0.500000f},    {0.000000f, -2.427051f, -0.500000f},
    {1.000000f, 0.809017f, 2.118034f},     {1.000000f, 0.809017f, -2.118034f},
    {1.000000f, -0.809017f, 2.118034f},    {1.000000f, -0.809017f, -2.118034f},
    {-1.000000f, 0.809017f, 2.118034f},    {-1.000000f, 0.809017f, -2.118034f},
    {-1.000000f, -0.809017f, 2.118034f},   {-1.000000f, -0.809017f, -2.118034f},
    {2.118034f, 1.000000f, 0.809017f},     {2.118034f, 1.000000f, -0.809017f},
    {2.118034f, -1.000000f, 0.809017f},    {2.118034f, -1.000000f, -0.809017f},
    {-2.118034f, 1.000000f, 0.809017f},    {-2.118034f, 1.000000f, -0.809017f},
    {-2.118034f, -1.000000f, 0.809017f},   {-2.118034f, -1.000000f, -0.809017f},
    {0.809017f, 2.118034f, 1.000000f},     {0.809017f, 2.118034f, -1.000000f},
    {0.809017f, -2.118034f, 1.000000f},    {0.809017f, -2.118034f, -1.000000f},
    {-0.809017f, 2.118034f, 1.000000f},    {-0.809017f, 2.118034f, -1.000000f},
    {-0.809017f, -2.118034f, 1.000000f},   {-0.809017f, -2.118034f, -1.000000f},
    {0.500000f, 1.618034f, 1.809017f},     {0.500000f, 1.618034f, -1.809017f},
    {0.500000f, -1.618034f, 1.809017f},    {0.500000f, -1.618034f, -1.809017f},
    {-0.500000f, 1.618034f, 1.809017f},    {-0.500000f, 1.618034f, -1.809017f},
    {-0.500000f, -1.618034f, 1.809017f},   {-0.500000f, -1.618034f, -1.809017f},
    {1.809017f, 0.500000f, 1.618034f},     {1.809017f, 0.500000f, -1.618034f},
    {1.809017f, -0.500000f, 1.618034f},    {1.809017f, -0.500000f, -1.618034f},
    {-1.809017f, 0.500000f, 1.618034f},    {-1.809017f, 0.500000f, -1.618034f},
    {-1.809017f, -0.500000f, 1.618034f},   {-1.809017f, -0.500000f, -1.618034f},
    {1.618034f, 1.809017f, 0.500000f},     {1.618034f, 1.809017f, -0.500000f},
    {1.618034f, -1.809017f, 0.500000f},    {1.618034f, -1.809017f, -0.500000f},
    {-1.618034f, 1.809017f, 0.500000f},    {-1.618034f, 1.809017f, -0.500000f},
    {-1.618034f, -1.809017f, 0.500000f},   {-1.618034f, -1.809017f, -0.500000f}
};
```



```c
    {-1.618034f, 1.809017f, 0.500000f},    {-1.618034f, 1.809017f, -0.500000f},
    {-1.618034f, -1.809017f, 0.500000f},   {-1.618034f, -1.809017f, -0.500000f}
};

// Soccer ball: 90 edges
const int soccer_edges[90][2] = {
    {0, 2}, {0, 12}, {0, 14},
    {1, 3}, {1, 13}, {1, 15},
    {2, 16}, {2, 18},
    {3, 17}, {3, 19},
    {4, 5}, {4, 20}, {4, 21},
    {5, 22}, {5, 23},
    {6, 7}, {6, 24}, {6, 25},
    {7, 26}, {7, 27},
    {8, 9}, {8, 28}, {8, 32},
    {9, 29}, {9, 33},
    {10, 11}, {10, 30}, {10, 34},
    {11, 31}, {11, 35},
    {12, 36}, {12, 44},
    {13, 37}, {13, 45},
    {14, 38}, {14, 46},
    {15, 39}, {15, 47},
    {16, 40}, {16, 48},
    {17, 41}, {17, 49},
    {18, 42}, {18, 50},
    {19, 43}, {19, 51},
    {20, 44}, {20, 52},
    {21, 45}, {21, 53},
    {22, 46}, {22, 54},
    {23, 47}, {23, 55},
```



```
    {23, 47}, {23, 55},
    {24, 48}, {24, 56},
    {25, 49}, {25, 57},
    {26, 50}, {26, 58},
    {27, 51}, {27, 59},
    {28, 36}, {28, 52},
    {29, 37}, {29, 53},
    {30, 38}, {30, 54},
    {31, 39}, {31, 55},
    {32, 40}, {32, 56},
    {33, 41}, {33, 57},
    {34, 42}, {34, 58},
    {35, 43}, {35, 59},
    {36, 40},
    {37, 41},
    {38, 42},
    {39, 43},
    {44, 46},
    {45, 47},
    {48, 50},
    {49, 51},
    {52, 53},
    {54, 55},
    {58, 59}
};
```

Vertices and edges of the truncated icosahedron (soccer ball shape) are defined as:

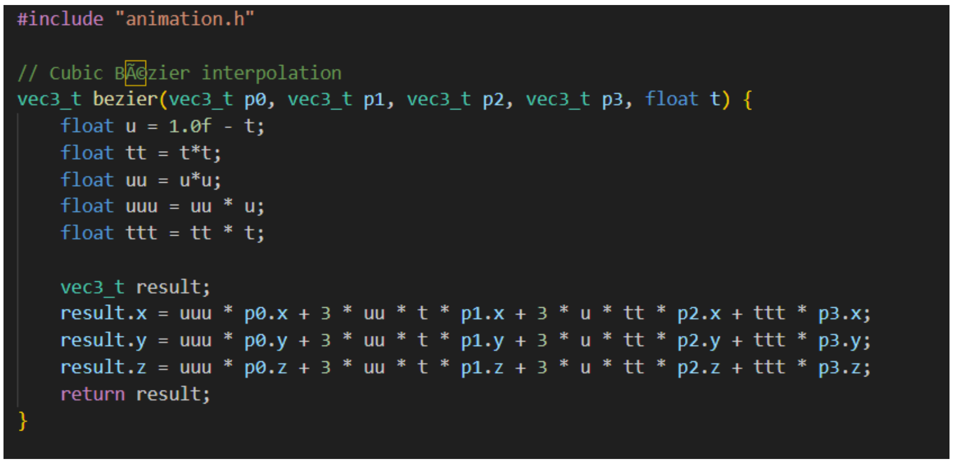
vec3_t soccer_vertices[60]

int soccer_edges[90][2]

This information defines the points and edge connections used to render the soccer ball structure.

<u>animation.c file:</u>

Bezier Animation – Smooth Position Interpolation

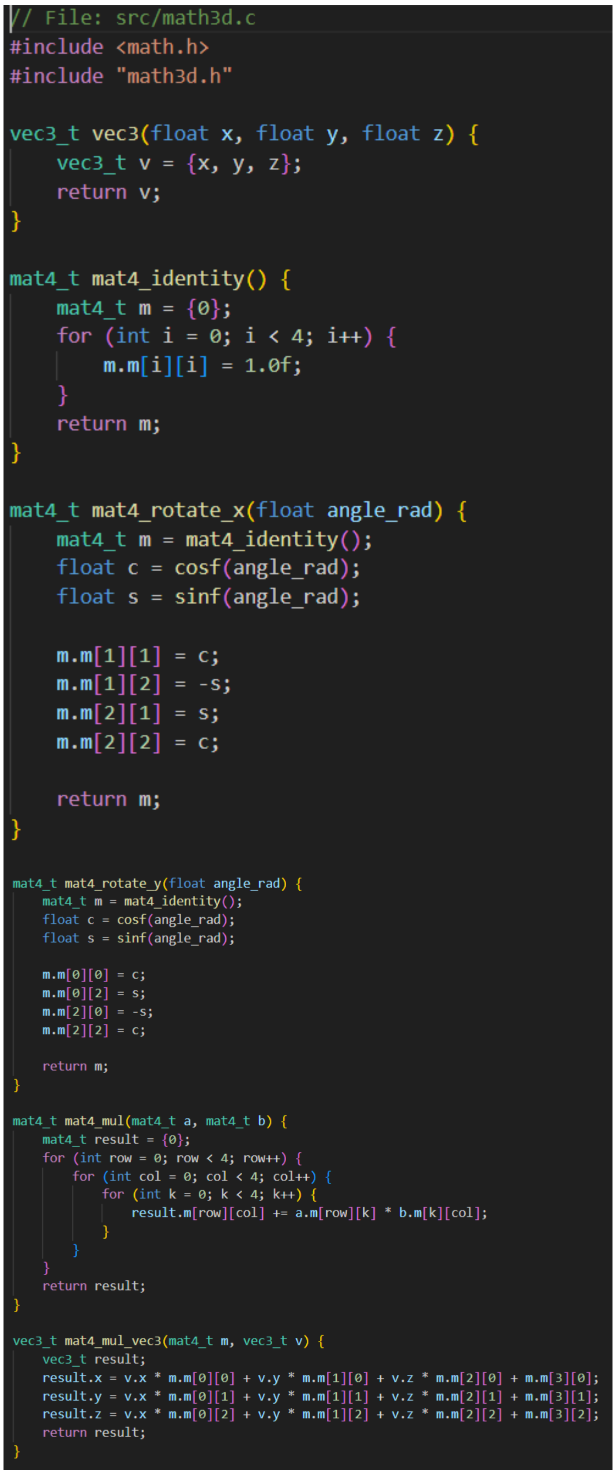


```
#include "animation.h"

// Cubic BÃ©zier interpolation
vec3_t bezier(vec3_t p0, vec3_t p1, vec3_t p2, vec3_t p3, float t) {
    float u = 1.0f - t;
    float tt = t*t;
    float uu = u*u;
    float uuu = uu * u;
    float ttt = tt * t;

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

- This function carries out cubic Bezier interpolation to animate the motion smoothly between the control points.
- Given four 3D points (p0 to p3) and a parameter t ∈ [0, 1], it produces the interpolated point.
- It gives natural, smooth, and cyclical motion to the 3D model.

<u>math3d.c file</u>

Matrix Operations

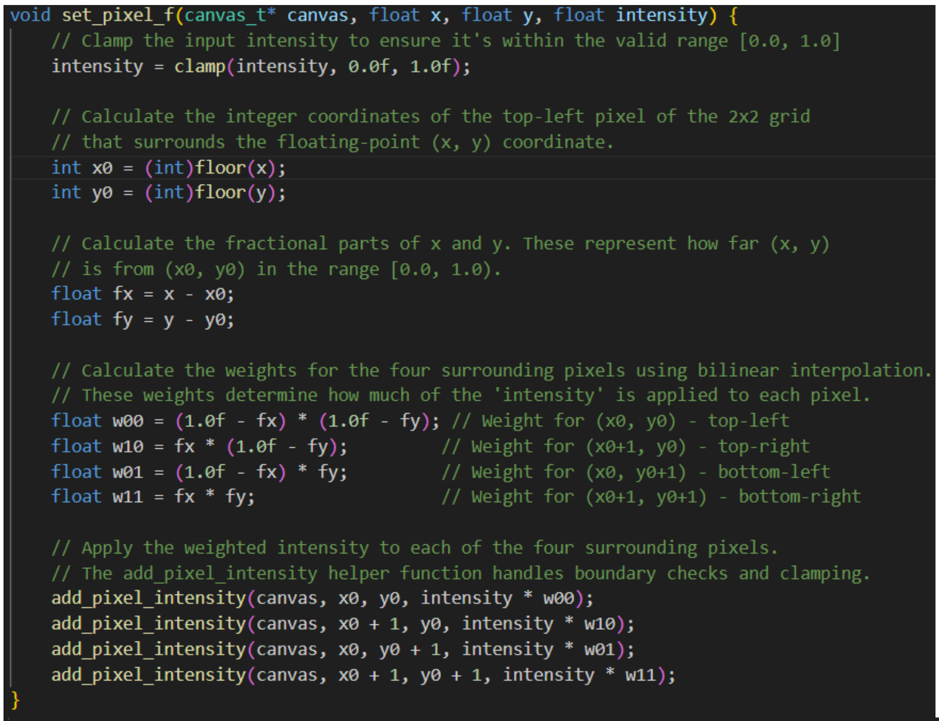


```c
// File: src/math3d.c
#include <math.h>
#include "math3d.h"

vec3_t vec3(float x, float y, float z) {
    vec3_t v = {x, y, z};
    return v;
}

mat4_t mat4_identity() {
    mat4_t m = {0};
    for (int i = 0; i < 4; i++) {
        m.m[i][i] = 1.0f;
    }
    return m;
}

mat4_t mat4_rotate_x(float angle_rad) {
    mat4_t m = mat4_identity();
    float c = cosf(angle_rad);
    float s = sinf(angle_rad);

    m.m[1][1] = c;
    m.m[1][2] = -s;
    m.m[2][1] = s;
    m.m[2][2] = c;

    return m;
}

mat4_t mat4_rotate_y(float angle_rad) {
    mat4_t m = mat4_identity();
    float c = cosf(angle_rad);
    float s = sinf(angle_rad);

    m.m[0][0] = c;
    m.m[0][2] = s;
    m.m[2][0] = -s;
    m.m[2][2] = c;

    return m;
}

mat4_t mat4_mul(mat4_t a, mat4_t b) {
    mat4_t result = {0};
    for (int row = 0; row < 4; row++) {
        for (int col = 0; col < 4; col++) {
            for (int k = 0; k < 4; k++) {
                result.m[row][col] += a.m[row][k] * b.m[k][col];
            }
        }
    }
    return result;
}

vec3_t mat4_mul_vec3(mat4_t m, vec3_t v) {
    vec3_t result;
    result.x = v.x * m.m[0][0] + v.y * m.m[1][0] + v.z * m.m[2][0] + m.m[3][0];
    result.y = v.x * m.m[0][1] + v.y * m.m[1][1] + v.z * m.m[2][1] + m.m[3][1];
    result.z = v.x * m.m[0][2] + v.y * m.m[1][2] + v.z * m.m[2][2] + m.m[3][2];
    return result;
}
```

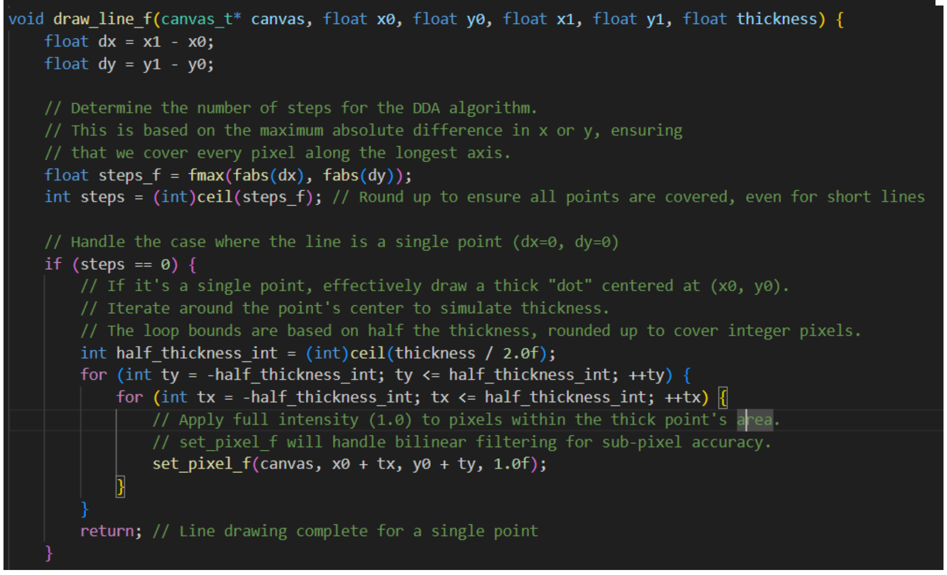
- mat4_rotate_x() and mat4_rotate_y() produce 4×4 rotation matrices.
- mat4_mul() multiplies these together to transform many times in succession.

canvas.c file

Line Drawing



```c
void set_pixel_f(canvas_t* canvas, float x, float y, float intensity) {
    // Clamp the input intensity to ensure it's within the valid range [0.0, 1.0]
    intensity = clamp(intensity, 0.0f, 1.0f);

    // Calculate the integer coordinates of the top-left pixel of the 2x2 grid
    // that surrounds the floating-point (x, y) coordinate.
    int x0 = (int)floor(x);
    int y0 = (int)floor(y);

    // Calculate the fractional parts of x and y. These represent how far (x, y)
    // is from (x0, y0) in the range [0.0, 1.0).
    float fx = x - x0;
    float fy = y - y0;

    // Calculate the weights for the four surrounding pixels using bilinear interpolation.
    // These weights determine how much of the 'intensity' is applied to each pixel.
    float w00 = (1.0f - fx) * (1.0f - fy); // Weight for (x0, y0) - top-left
    float w10 = fx * (1.0f - fy);         // Weight for (x0+1, y0) - top-right
    float w01 = (1.0f - fx) * fy;         // Weight for (x0, y0+1) - bottom-left
    float w11 = fx * fy;                  // Weight for (x0+1, y0+1) - bottom-right

    // Apply the weighted intensity to each of the four surrounding pixels.
    // The add_pixel_intensity helper function handles boundary checks and clamping.
    add_pixel_intensity(canvas, x0, y0, intensity * w00);
    add_pixel_intensity(canvas, x0 + 1, y0, intensity * w10);
    add_pixel_intensity(canvas, x0, y0 + 1, intensity * w01);
    add_pixel_intensity(canvas, x0 + 1, y0 + 1, intensity * w11);
}
```

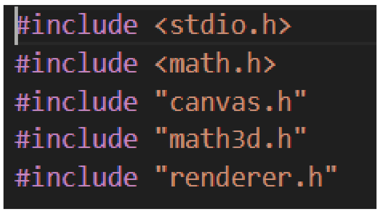


```c
void draw_line_f(canvas_t* canvas, float x0, float y0, float x1, float y1, float thickness) {
    float dx = x1 - x0;
    float dy = y1 - y0;

    // Determine the number of steps for the DDA algorithm.
    // This is based on the maximum absolute difference in x or y, ensuring
    // that we cover every pixel along the longest axis.
    float steps_f = fmax(fabs(dx), fabs(dy));
    int steps = (int)ceil(steps_f); // Round up to ensure all points are covered, even for short lines

    // Handle the case where the line is a single point (dx=0, dy=0)
    if (steps == 0) {
        // If it's a single point, effectively draw a thick "dot" centered at (x0, y0).
        // Iterate around the point's center to simulate thickness.
        // The loop bounds are based on half the thickness, rounded up to cover integer pixels.
        int half_thickness_int = (int)ceil(thickness / 2.0f);
        for (int ty = -half_thickness_int; ty <= half_thickness_int; ++ty) {
            for (int tx = -half_thickness_int; tx <= half_thickness_int; ++tx) {
                // Apply full intensity (1.0) to pixels within the thick point's area.
                // set_pixel_f will handle bilinear filtering for sub-pixel accuracy.
                set_pixel_f(canvas, x0 + tx, y0 + ty, 1.0f);
            }
        }
        return; // Line drawing complete for a single point
    }
```

- draw_line_f() draws thick, anti-aliased lines using DDA.

- set_pixel_f() performs bilinear filtering to spread brightness smoothly.

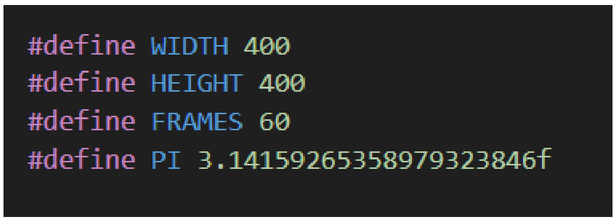
File soccer_demo.c



```
#include <stdio.h>
#include <math.h>
#include "canvas.h"
#include "math3d.h"
#include "renderer.h"
```

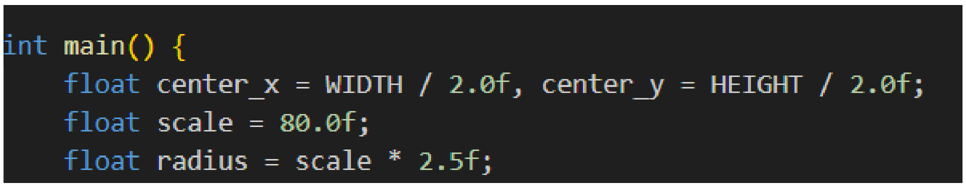
These are the headers:

- Standard I/O (stdio.h) and math functions (math.h),
- Special modules for rendering on a canvas (canvas.h),
- 3D math functions (math3d.h), and
- Rendering routines and mesh information (renderer.h).



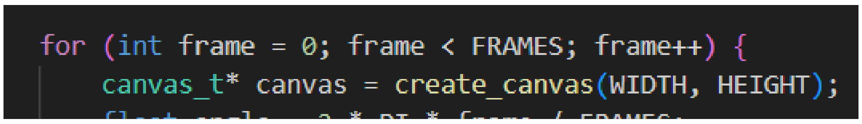
```
#define WIDTH 400
#define HEIGHT 400
#define FRAMES 60
#define PI 3.14159265358979323846f
```

These constants define the canvas size (400x400 pixels), number of animation frames (60), and the mathematical constant π for rotation calculation.



```
int main() {
    float center_x = WIDTH / 2.0f, center_y = HEIGHT / 2.0f;
    float scale = 80.0f;
    float radius = scale * 2.5f;
```

We compute the canvas center and determine a scale factor to project 3D coordinates onto 2D. radius sets the clipping range with a circular viewport.



```
  for (int frame = 0; frame < FRAMES; frame++) {
      canvas_t* canvas = create_canvas(WIDTH, HEIGHT);
```

This loop generates 60 frames. For each frame, an empty new canvas is created.

```c
float angle = 2 * PI * frame / FRAMES;
mat4_t rot_y = mat4_rotate_y(angle);
mat4_t rot_x = mat4_rotate_x(angle / 2);
mat4_t transform = mat4_mul(rot_y, rot_x);
```

We compute a time-varying angle to rotate the soccer ball. It's rotated about the Y-axis (horizontal rotation) and X-axis (tilt), giving its dynamic 3D rotation. Both of these matrices are then multiplied together in order to form a combined transformation.

```c
render_wireframe(
    canvas,
    soccer_vertices,
    soccer_edges,
    60,
    90,
    transform,
    center_x,
    center_y,
    scale,
    radius
);
```

This function projects the transformed 3D soccer ball onto the 2D canvas by using the provided transform matrix and draws its wireframe, clipping it to the circular viewport.

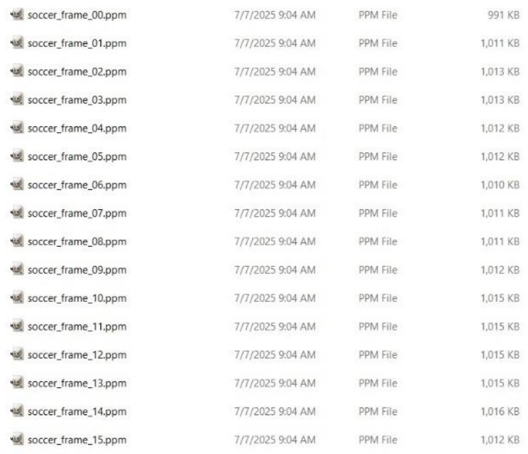
```c
        char filename[64];
        snprintf(filename, sizeof(filename), "soccer_frame_%02d.ppm", frame);
        save_canvas_to_ppm(canvas, filename);
        destroy_canvas(canvas);
    }
    return 0;
}
```

Each rendered frame is saved as a .ppm image (soccer_frame_00.ppm, ., soccer_frame_59.ppm). The canvas is then released to free memory.

Output

In this task, we demonstrated the 3D rendering pipeline by rotating a truncated icosahedron (soccer ball) and projecting onto a 2D canvas. The pipeline of transformation includes rotation around Y-axis and X-axis, and perspective projection clipped to circular viewport.

60 frames were generated with the following output file names:



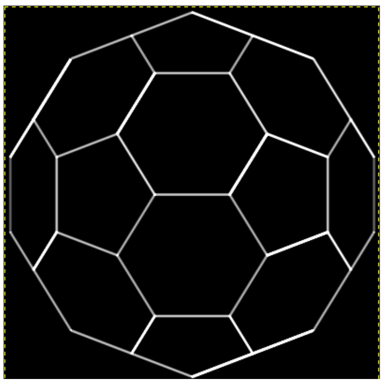
```
soccer_frame_00.ppm    7/7/2025 9:04 AM    PPM File      991 KB
soccer_frame_01.ppm    7/7/2025 9:04 AM    PPM File    1,011 KB
soccer_frame_02.ppm    7/7/2025 9:04 AM    PPM File    1,013 KB
soccer_frame_03.ppm    7/7/2025 9:04 AM    PPM File    1,013 KB
soccer_frame_04.ppm    7/7/2025 9:04 AM    PPM File    1,012 KB
soccer_frame_05.ppm    7/7/2025 9:04 AM    PPM File    1,012 KB
soccer_frame_06.ppm    7/7/2025 9:04 AM    PPM File    1,010 KB
soccer_frame_07.ppm    7/7/2025 9:04 AM    PPM File    1,011 KB
soccer_frame_08.ppm    7/7/2025 9:04 AM    PPM File    1,011 KB
soccer_frame_09.ppm    7/7/2025 9:04 AM    PPM File    1,012 KB
soccer_frame_10.ppm    7/7/2025 9:04 AM    PPM File    1,015 KB
soccer_frame_11.ppm    7/7/2025 9:04 AM    PPM File    1,015 KB
soccer_frame_12.ppm    7/7/2025 9:04 AM    PPM File    1,015 KB
soccer_frame_13.ppm    7/7/2025 9:04 AM    PPM File    1,015 KB
soccer_frame_14.ppm    7/7/2025 9:04 AM    PPM File    1,016 KB
soccer_frame_15.ppm    7/7/2025 9:04 AM    PPM File    1,012 KB
```

Each picture was drawn exactly at resolution 400 × 400, and a circular clipping region was specified to display the rotating soccer ball with clean wireframe.

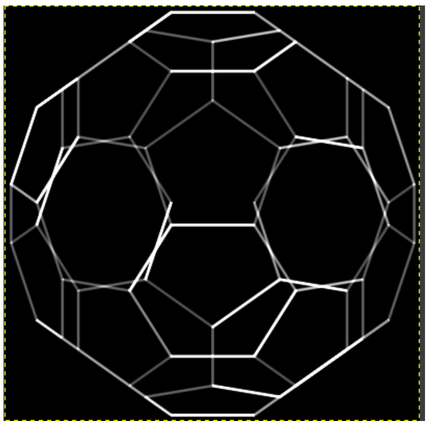
Sample Output Frames

Below are sample snapshots from the output frames produced to demonstrate the animation:

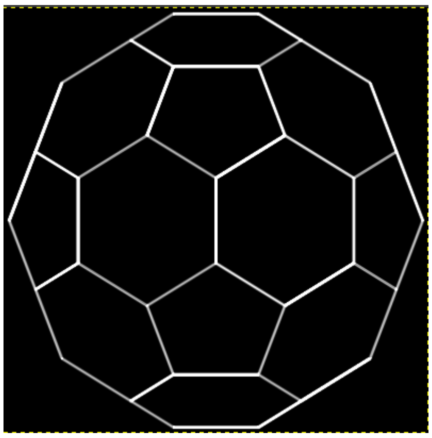
- Frame 0



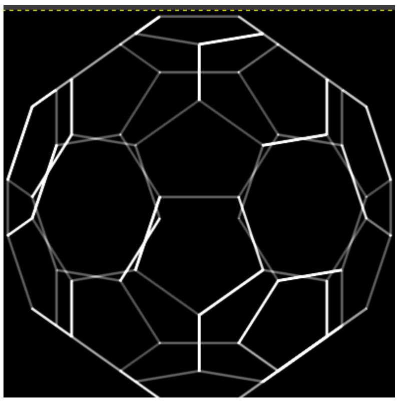
- Frame 15



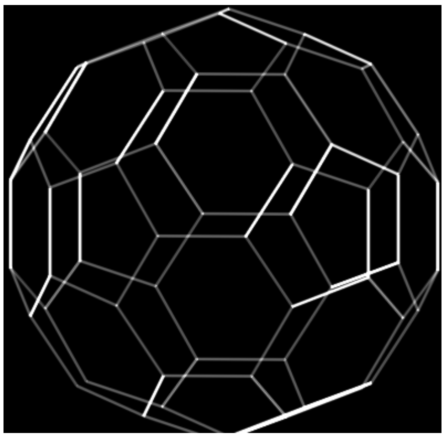
- Frame 30



- Frame 45



- Frame 59



The more than 60 frames drawn of wireframe rotating soccer ball confirm the following:

- Correctness of 3D transformation via mat4_rotate_x() and mat4_rotate_y().
- Correct projection and clipping via render_wireframe() and circular viewport code.
- Smooth line drawing with sub-pixel accuracy with draw_line_f().

Task 4: Lighting & Polish

Purpose

Purpose of Task 3.4 is to enhance the visual realism of 3D objects rendered from wireframes in our software renderer (libtiny3d) by the inclusion of:

- Lambertian lighting for realistic directional brightness.
- Cubic Bézier animations for smooth, curvy movement of objects.
- Synchronization and looping, allowing for a number of shapes to animate in unison over a loop.

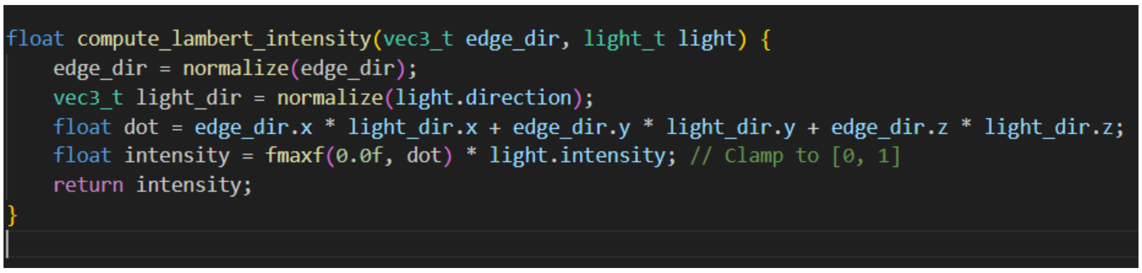
This task brings polish and beauty to the final demo.

Design

1. Lambert Lighting:
    - Uses dot product of light direction and edge direction to determine brightness.
    - Bright edges are only those towards the light.
    - Dynamic lighting employs several light sources.
2. Cubic Bézier Curve Animation:
    - Smooth curved path is defined by four control points.
    - A time parameter t (0 to 1) interpolates along the curve.
    - Supports continuous loops of animation.
3. Sync & Looping:
    - Time is split evenly for soccer ball and cube paths.
    - The motion cycles every 60 frames.
    - Ensures multiple objects animate together in phase.

Implementation

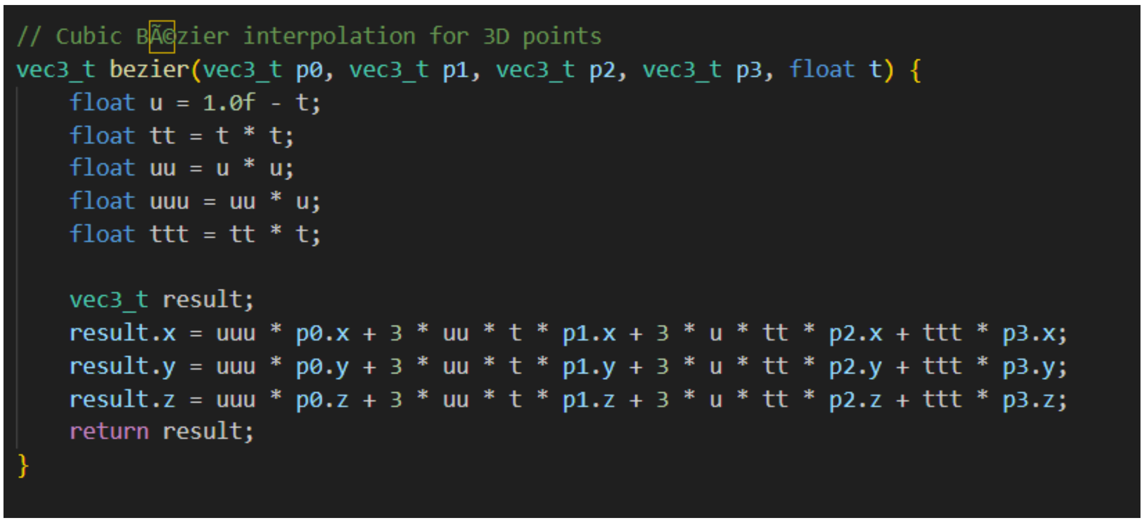
File lighting.c – Lambertian Lighting



```c
float compute_lambert_intensity(vec3_t edge_dir, light_t light) {
    edge_dir = normalize(edge_dir);
    vec3_t light_dir = normalize(light.direction);
    float dot = edge_dir.x * light_dir.x + edge_dir.y * light_dir.y + edge_dir.z * light_dir.z;
    float intensity = fmaxf(0.0f, dot) * light.intensity; // Clamp to [0, 1]
    return intensity;
}
```

- Computes the extent to which a line is "oriented" toward the light.
- Normalizes the two vectors.
- Computes the dot product.
- Clamps to [0, 1] to avoid negative lighting.
- Multiples by the factor of the light's intensity.

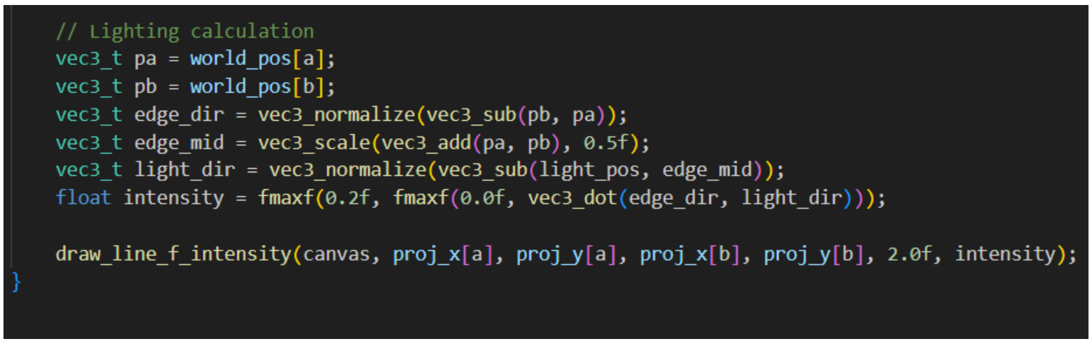
File animation.c – Bézier Curve



```c
// Cubic BÃ©zier interpolation for 3D points
vec3_t bezier(vec3_t p0, vec3_t p1, vec3_t p2, vec3_t p3, float t) {
    float u = 1.0f - t;
    float tt = t * t;
    float uu = u * u;
    float uuu = uu * u;
    float ttt = tt * t;

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

- Computes a point on a cubic Bézier curve when parameter t is known.
- Interpolates the weighted contribution of 4 control points.
- Delivers a smooth motion path starting at p0 and ending in p3.

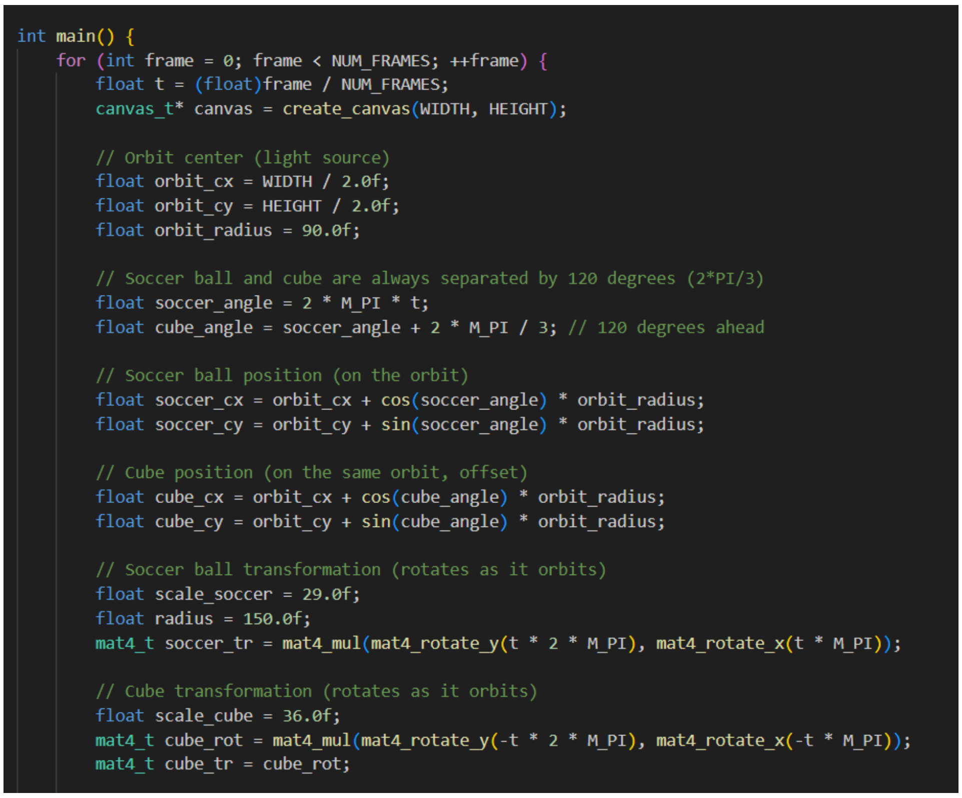
File render_wireframe() – Better Wireframe Drawing



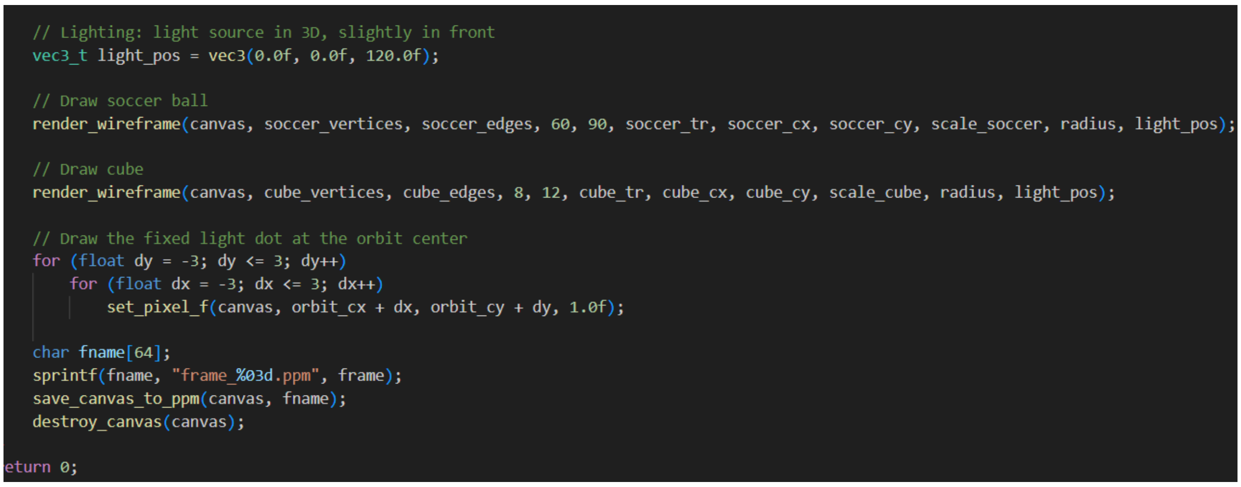
```c
    // Lighting calculation
    vec3_t pa = world_pos[a];
    vec3_t pb = world_pos[b];
    vec3_t edge_dir = vec3_normalize(vec3_sub(pb, pa));
    vec3_t edge_mid = vec3_scale(vec3_add(pa, pb), 0.5f);
    vec3_t light_dir = vec3_normalize(vec3_sub(light_pos, edge_mid));
    float intensity = fmaxf(0.2f, fmaxf(0.0f, vec3_dot(edge_dir, light_dir)));

    draw_line_f_intensity(canvas, proj_x[a], proj_y[a], proj_x[b], proj_y[b], 2.0f, intensity);
}
```

- Computes direction and midpoint of each edge.
- Computes direction from light to edge.
- Uses dot product to compute brightness.
- Applies each brightness to each line with draw_line_f_intensity.

main.c – The Animated Demo



```c
int main() {
    for (int frame = 0; frame < NUM_FRAMES; ++frame) {
        float t = (float)frame / NUM_FRAMES;
        canvas_t* canvas = create_canvas(WIDTH, HEIGHT);

        // Orbit center (light source)
        float orbit_cx = WIDTH / 2.0f;
        float orbit_cy = HEIGHT / 2.0f;
        float orbit_radius = 90.0f;

        // Soccer ball and cube are always separated by 120 degrees (2*PI/3)
        float soccer_angle = 2 * M_PI * t;
        float cube_angle = soccer_angle + 2 * M_PI / 3; // 120 degrees ahead

        // Soccer ball position (on the orbit)
        float soccer_cx = orbit_cx + cos(soccer_angle) * orbit_radius;
        float soccer_cy = orbit_cy + sin(soccer_angle) * orbit_radius;

        // Cube position (on the same orbit, offset)
        float cube_cx = orbit_cx + cos(cube_angle) * orbit_radius;
        float cube_cy = orbit_cy + sin(cube_angle) * orbit_radius;

        // Soccer ball transformation (rotates as it orbits)
        float scale_soccer = 29.0f;
        float radius = 150.0f;
        mat4_t soccer_tr = mat4_mul(mat4_rotate_y(t * 2 * M_PI), mat4_rotate_x(t * M_PI));

        // Cube transformation (rotates as it orbits)
        float scale_cube = 36.0f;
        mat4_t cube_rot = mat4_mul(mat4_rotate_y(-t * 2 * M_PI), mat4_rotate_x(-t * M_PI));
        mat4_t cube_tr = cube_rot;
```

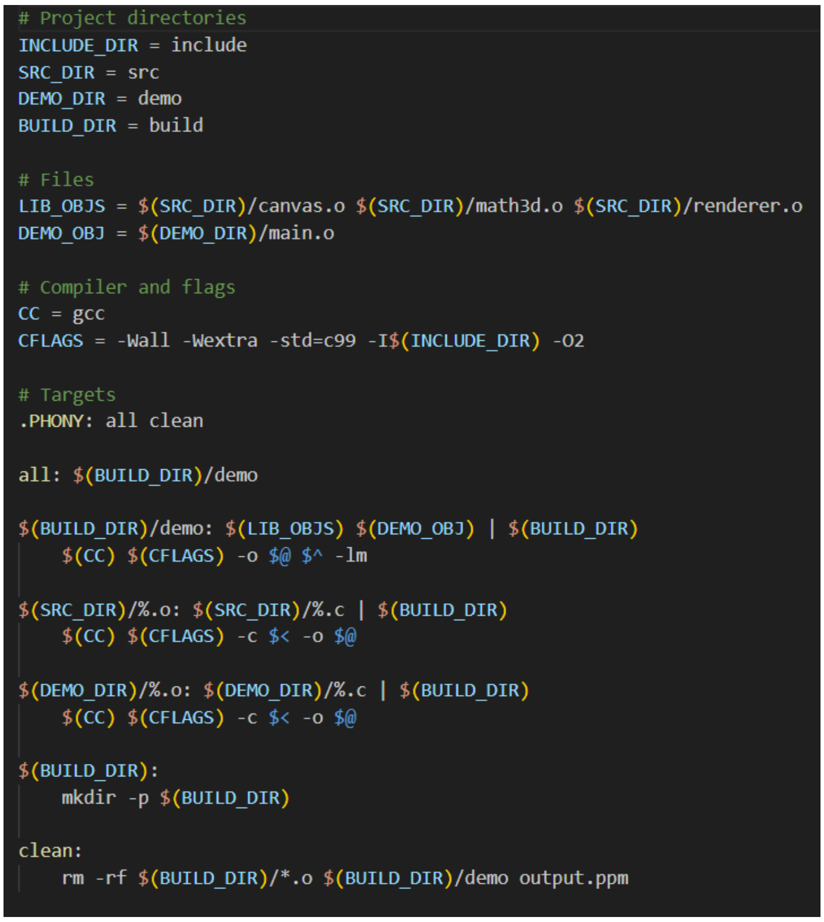


```c
    // Lighting: light source in 3D, slightly in front
    vec3_t light_pos = vec3(0.0f, 0.0f, 120.0f);

    // Draw soccer ball
    render_wireframe(canvas, soccer_vertices, soccer_edges, 60, 90, soccer_tr, soccer_cx, soccer_cy, scale_soccer, radius, light_pos);

    // Draw cube
    render_wireframe(canvas, cube_vertices, cube_edges, 8, 12, cube_tr, cube_cx, cube_cy, scale_cube, radius, light_pos);

    // Draw the fixed light dot at the orbit center
    for (float dy = -3; dy <= 3; dy++)
        for (float dx = -3; dx <= 3; dx++)
            set_pixel_f(canvas, orbit_cx + dx, orbit_cy + dy, 1.0f);

    char fname[64];
    sprintf(fname, "frame_%03d.ppm", frame);
    save_canvas_to_ppm(canvas, fname);
    destroy_canvas(canvas);

eturn 0;
```

- Soccer ball and cube animated using time parameter t.
- Light position set to z = 120.
- Independent transforms for ball and cube, both rotating about center.
- Lighting and movement both updated each frame.

Compilation & Execution

Makefile



```make
# Project directories
INCLUDE_DIR = include
SRC_DIR = src
DEMO_DIR = demo
BUILD_DIR = build

# Files
LIB_OBJS = $(SRC_DIR)/canvas.o $(SRC_DIR)/math3d.o $(SRC_DIR)/renderer.o
DEMO_OBJ = $(DEMO_DIR)/main.o

# Compiler and flags
CC = gcc
CFLAGS = -Wall -Wextra -std=c99 -I$(INCLUDE_DIR) -O2

# Targets
.PHONY: all clean

all: $(BUILD_DIR)/demo

$(BUILD_DIR)/demo: $(LIB_OBJS) $(DEMO_OBJ) | $(BUILD_DIR)
	$(CC) $(CFLAGS) -o $@ $^ -lm

$(SRC_DIR)/%.o: $(SRC_DIR)/%.c | $(BUILD_DIR)
	$(CC) $(CFLAGS) -c $< -o $@

$(DEMO_DIR)/%.o: $(DEMO_DIR)/%.c | $(BUILD_DIR)
	$(CC) $(CFLAGS) -c $< -o $@

$(BUILD_DIR):
	mkdir -p $(BUILD_DIR)

clean:
	rm -rf $(BUILD_DIR)/*.o $(BUILD_DIR)/demo output.ppm
```

- To compile and run:
  ```
  make clean
  make
  ./build/demo
  ```

- This generates 60 .ppm frames (e.g., frame_000.ppm through frame_059.ppm).

Output

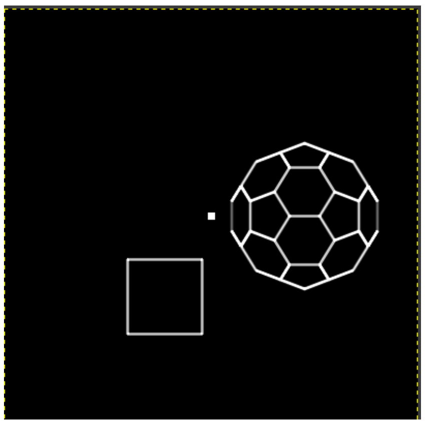
The final product of Task 3.4 demonstrates the mature graphical ability of the libtiny3d renderer. In 60 frames, two 3D wireframe models—a cube and a soccer ball-orbit a point while rotating and responding dynamically to lighting.

Each frame records:

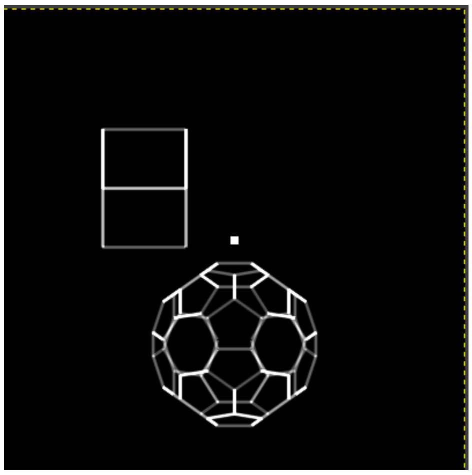
- Real-time Lambertian shading—edges towards the light source being brighter.
- Smooth and organic Bézier-curve-based animation for both the cube and the soccer ball.
- Strong synchronization and looping—the animation smoothly loops back after a full iteration.

Following are some sample screenshots from the produced frames:

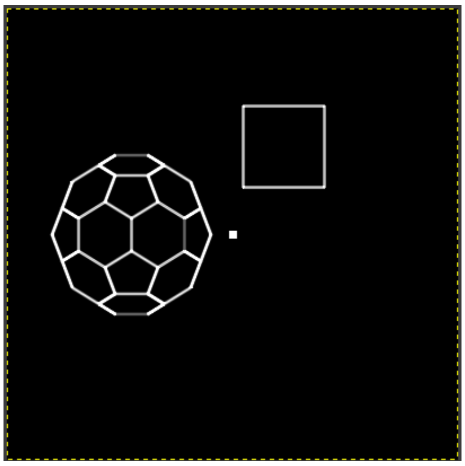
- Frame 0: Initial Position



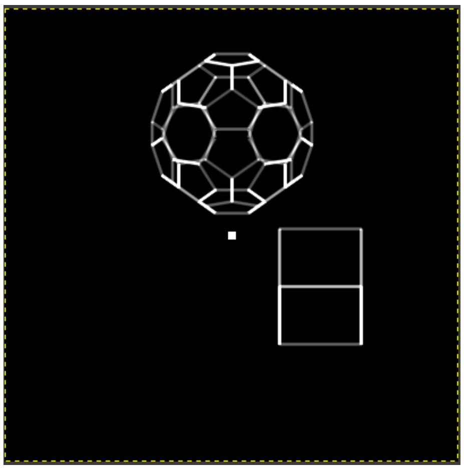
- Frame 15: Quarter Rotation



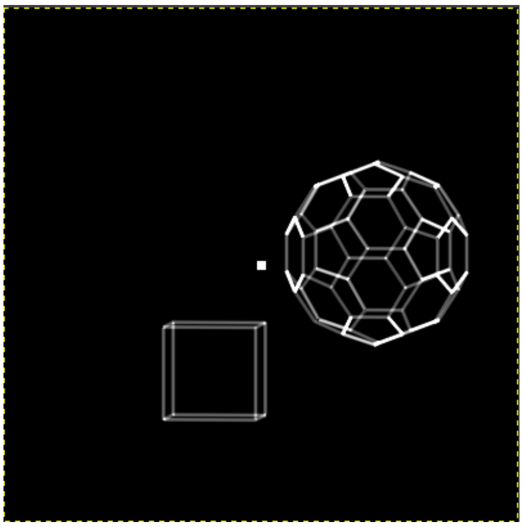
- Frame 30: Half Rotation



- Frame 45: Three-Quarter Rotation



- Frame 59: End of Loop



These output frames clearly confirm successful integration of lighting, animation, synchronization, and smooth rendering—showing completion of a smooth and dynamic 3D rendering pipeline in the libtiny3d project.

4. Design Decisions & Assumptions

In the process of developing libtiny3d, we made some significant design decisions for simplicity, performance, and quality of appearance. All decisions were guided by the goals of the project as well as the nature of 3D rendering.

DDA Instead of Bresenham

We used the Digital Differential Analyzer (DDA) line-drawing algorithm instead of Bresenham's. Since our canvas is float-point based coordinates, DDA is a better fit and less likely to need integer conversion. This left our code cleaner and more suitable for smooth, sub-pixel rendering.

Flat, Column-Major Matrices

Our mat4_t class uses a flat column-major 1D array of 16 floats. This is compliant with graphics APIs like OpenGL and allows for more efficient memory access. It also sets the code up for future expansion of GPU-based rendering.

Bilinear Filtering for Smooth Lines

To improve visual quality, we used bilinear filtering in set_pixel_f() rather than rounding coordinates. This smooths intensity across adjacent pixels and avoids jagged edges, producing extremely much smoother lines, especially for diagonal strokes.

Circular Clipping & Fixed Animation Steps

- Circular viewport clipping was used to keep the soccer ball demo well bounded and visually centered.
- In order to render animations, we used a fixed time step to allow synchronized and uniform motion. Variable steps offer more freedom, but the fixed rate made development and debugging simpler.

These decisions allowed us to implement a renderer that's visually strong, optimal, and easy to develop-attaining technical and artistic goals of our project.

5. Mathematical Explanations

libtiny3d's success relies on core mathematical principles. Here's how we applied them to render 3D objects on a 2D screen:

Bilinear Filtering

We draw smooth lines with bilinear filtering in set_pixel_f(). When we're between four grid points (due to floating-point coordinates), we interpolate the intensity between the closest four pixels. This avoids blocky or jagged edges and gives us smooth, anti-aliased lines.

Subpixel intensity at position (x, y) is computed in two steps:

- Horizontally interpolate between the left and right values first.
- Then interpolate between the top and bottom accordingly.

This gives us nice-looking gradients, especially important for lines and curves.

Fast Inverse Square Root (Vector Normalization)

Normalizing a vector typically involves a square root operation, which is expensive. We utilized the well-known fast inverse square root trick (popularized by Quake III Arena) to get an optimized estimate of 1/sqrt(x) using bit twiddling and Newton's method.

For a vector v = (x, y, z):

- First compute the squared length: $len^2 = x^2 + y^2 + z^2$
- Then use the fast inverse square root to approximate $1/sqrt(len^2)$
- Finally, multiply every component of the vector by this value to normalize it

This method is fast and good enough for real-time 3D rendering.

Smooth Interpolation: SLERP & Bézier

SLERP (Spherical Linear Interpolation)

In order to rotate smoothly between two directions, we use SLERP. It's a curve interpolation between two vectors a and b on the surface of a sphere. It stops abrupt movements and provides smooth natural rotations (like a turn of a camera or a rotating object).

Bézier Curves (Cubic)

To achieve more artistic animations, we employed cubic Bézier curves. There are four control points (P0–P3) that define a curve, and the position at time t is determined by a weighted sum:

$B(t) = (1-t)^3 P0 + 3(1-t)^2 t\, P1 + 3(1-t)t^2 P2 + t^3 P3$

Smooth, flowing curves are what we achieve, ideal for animating moving objects or camera transitions.

Matrix Transformations

We utilized 4×4 matrices to carry out all transformations of 3D objects:

- Translation (move):

$$\begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Scaling (resize):

$$\begin{pmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Rotation (e.g., around Z axis):

$$R_z(\theta) = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

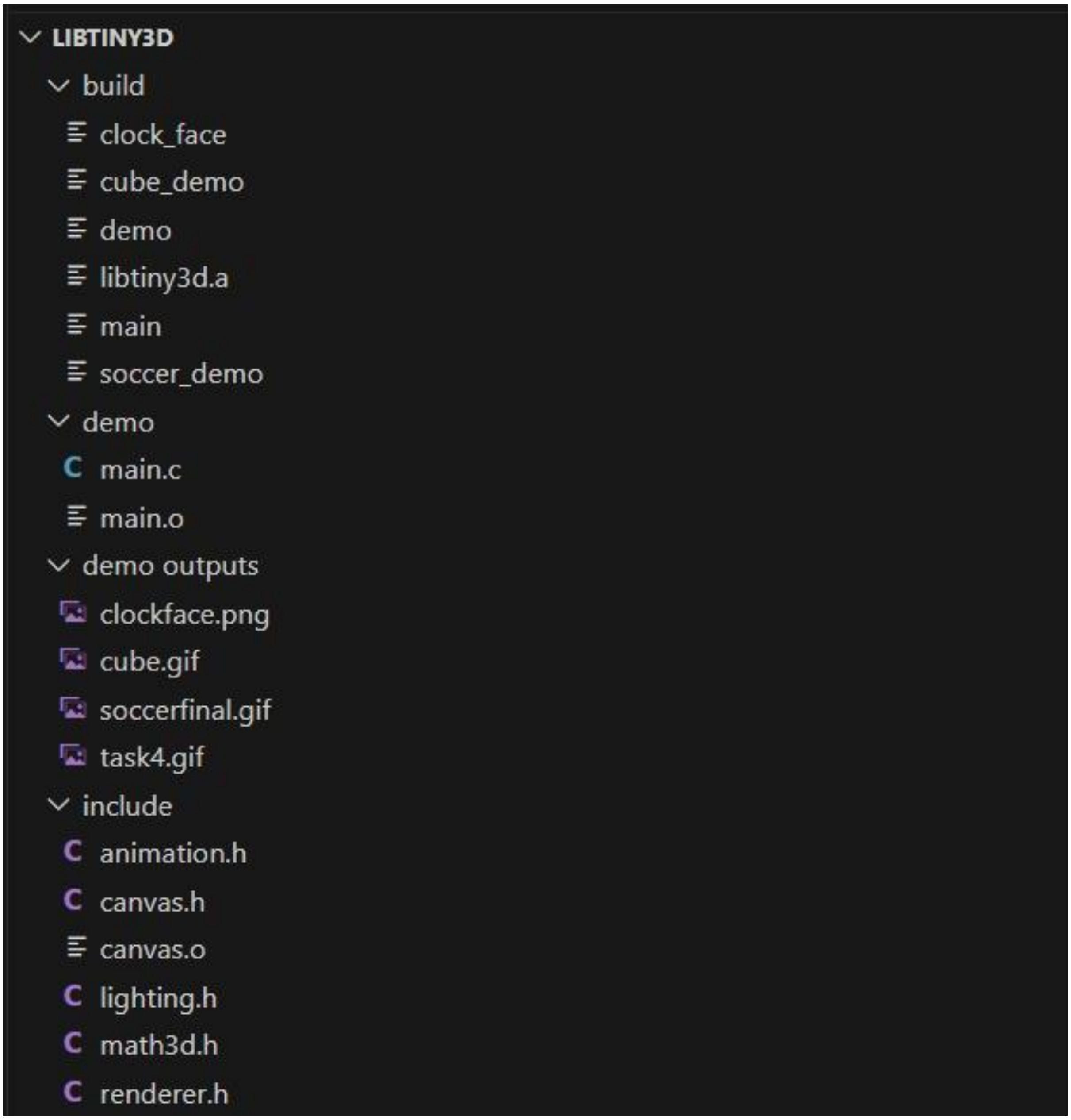
There are similar ones for X and Y axes.

- Perspective Projection (Frustum):
  This matrix projects 3D points into 2D space with perspective for depth. Our skew frustum matrix enables us to control left, right, top, bottom, near, and far clipping planes, and this gives us dynamic camera perspectives.

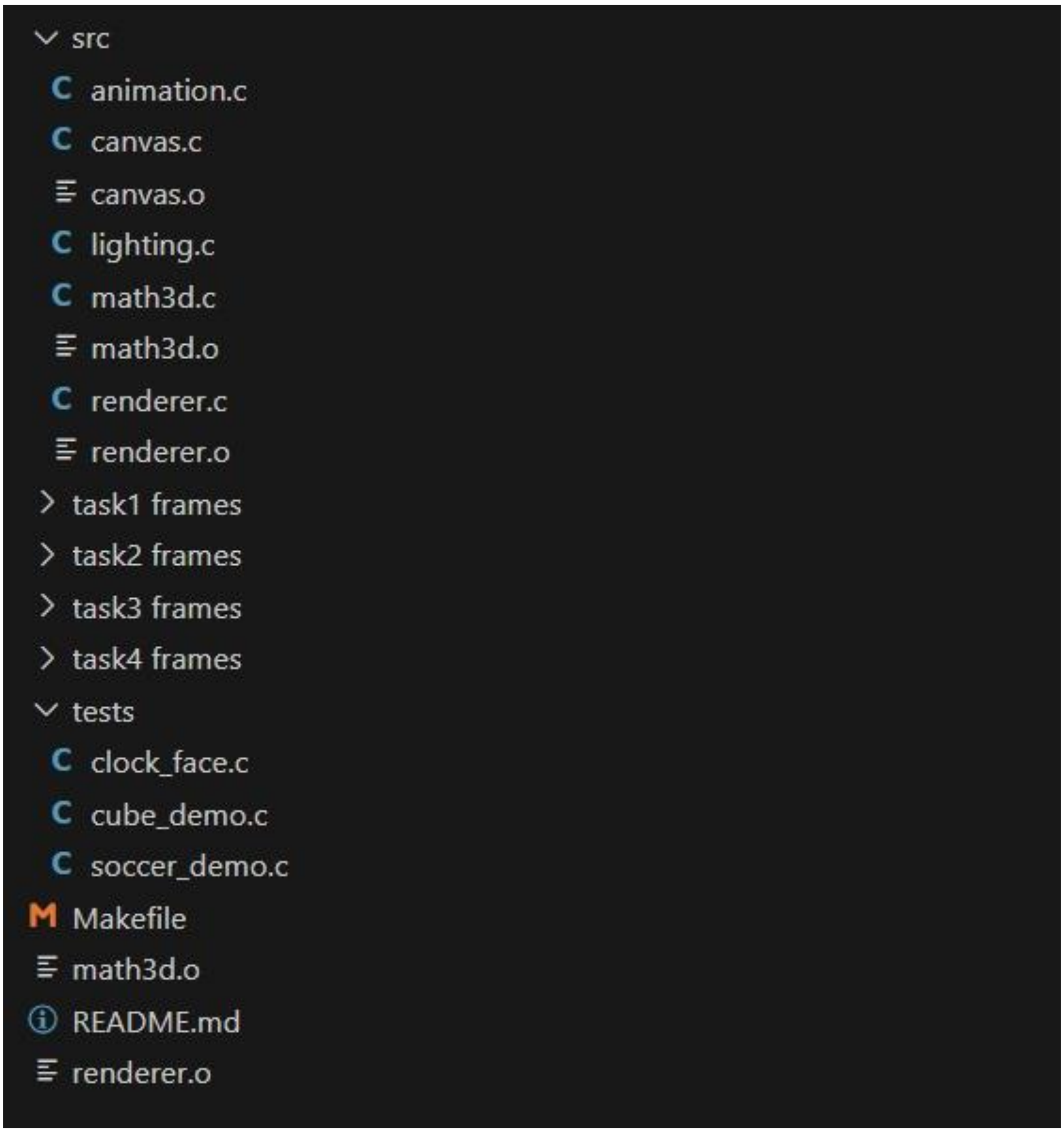
These matrices are all put together into one in a specific order (Model → View → Projection) to project 3D objects into their final 2D screen coordinates.

6. Project Structure

Our libtiny3d project is structured in a clean and rational directory hierarchy to ease development, testing, and documentation.



```
LIBTINY3D
  build
    clock_face
    cube_demo
    demo
    libtiny3d.a
    main
    soccer_demo
  demo
    main.c
    main.o
  demo outputs
    clockface.png
    cube.gif
    soccerfinal.gif
    task4.gif
  include
    animation.h
    canvas.h
    canvas.o
    lighting.h
    math3d.h
    renderer.h
```



```
∨ src
    animation.c
    canvas.c
    canvas.o
    lighting.c
    math3d.c
    math3d.o
    renderer.c
    renderer.o
  > task1 frames
  > task2 frames
  > task3 frames
  > task4 frames
∨ tests
    clock_face.c
    cube_demo.c
    soccer_demo.c
  Makefile
  math3d.o
  README.md
  renderer.o
```

7. Demo Overview

Our demo video is a full visual tour of the main features and functionality realized in the libtiny3d project. It shows the development process step by step, starting from our basic graphics engine and transforming it into a full-fledged wireframe renderer with animation and lighting.

1. Drawing Radial Lines on Canvas

We begin with the simplest operation-drawing lines. Employing the canvas_t structure and the draw_line_f() routine, the demo demonstrates a clock-like pattern of radial lines being drawn from the center of the canvas. This exercise is meant to demonstrate how bilinear filtering and the DDA algorithm facilitate smooth sub-pixel rendering, yielding high-quality, anti-aliased lines.

2. Transforming and Rotating a Cube

Then the demo highlights the math engine in 3D. A wireframe cube is displayed and rotated with custom matrix transformations (mat4_t). We perform a mix of rotations around the X and Y axes, visually checking that our matrix operations-such as mat4_rotate_x(), mat4_rotate_y(), and mat4_mul()-function properly. Here, we show that we are able to transform objects in 3D space and project them correctly onto a 2D canvas.

3. Soccer Ball Rendering with Viewport Clipping

Having tried simple 3D rendering using the cube, we move on to advanced by incorporating a truncated icosahedron-the shape of a soccer ball. Such a complex mesh is rendered by invoking the render_wireframe() method. Smooth rotation of the ball and circular clipping keep it snugly packed in a specified viewport. This scene illustrates that our renderer can handle more advanced models with proper clipping and transformation.

4. Lit and Animated Wireframes

The last of these is the most graphically packed. It employs animation from Bezier curves, Lambert lighting, and synchronized movement of multiple wireframe objects. The soccer ball and cube travel about a central point along flowing curves defined by cubic Bézier curves. Lighting intensity is dynamically calculated based on each edge's direction relative to the light source using the dot product method (compute_lambert_intensity()).

As the scene continues, objects move as a group, nicely turn, and their edges light up as per their rotation-creating an elegant, active, and realistic wireframe animation.

8. AI Tools Usage

In an effort to enhance the organization, readability, and quality of our lab reports, we utilized OpenAI's ChatGPT as a collaborative tool throughout the project. The AI operated largely as a collaborator in writing and debugging, helping us with the following specific tasks:

- Code Explanation & Documentation
  We utilized ChatGPT to:
  Describe how every component of our code (e.g., matrix calculations, light logic, Bezier interpolation) worked.Restate technical jargon in human-readable and report-compliant form.Condense long code segments into structured explanations for the lab report.

- Debugging Tips
  During development, the AI helped:
  Identify the cause of compilation/linking errors such as inconsistent function signatures or unresolved references.Make suggestions for organizing the Makefile, and module-izing code through header/source

separation.Release AI-trained models to better manage their memory usage during runtime.

- Report Structuring & Proofreading
  We also used AI to:
  Organize the lab report into clearly discernible sections (Purpose, Design, Implementation, Output, etc.).
  Develop humanized task overviews, demo actions, and issues faced.
  Improve grammar and readability in our final written report.

- Limitations
  AI was not utilized to write or develop the actual rendering code algorithms or matrix mathematics functions.

All of the code logic, transformations, lighting, and animation was developed, tested, and debugged by us with AI only explaining or providing alternative words when needed.

9. Individual Contributions

Our libtiny3d project success was achieved through efficient collaboration and good sharing of tasks. Each of us contributed our own strengths to ensure easy progress in all tasks. Here's how we individually contributed:

H.C.V. Perera- E/22/280

Chamudi spent most of her time on Task 1 (Canvas & Line Drawing) and Task 2 (3D Math Foundation). She implemented low-level canvas logic, including bilinear pixel rendering and smooth line drawing using the DDA algorithm. She also implemented vector and matrix math functions that served as the backbone of the 3D engine. These were used as the basis for all rendering and transformation tasks.

M.D.S. Senanayake- E/22/366

Senara contributed to the more advanced parts of the project-Task 3 (3D Rendering Pipeline) and Task 4 (Lighting & Polish). He put into action the base rendering loop render_wireframe(), handled clipping logic, and integrated Lambert lighting for true edge shading. Senara also developed the animation system using cubic Bézier curves, allowing synchronized object translation and looping.

Both of Us

Both of us worked together on:

- Designing and structuring the overall libtiny3d framework.
- Making the Makefile and managing compilation and linking.
- Making the lab report, so that design, code, and results for each task were well explained.
- Recording and post-processing the demo video, which graphically showcases all critical functionalities of the project.

Together, our combined efforts and with close collaboration, we were able to develop a complete 3D rendering engine from scratch and successfully demonstrate it.

10.Challenges Faced & Lessons Learned

Developing libtiny3d was a frustrating but very rewarding experience. Throughout the project, we were faced with so many technical challenges that tested our understanding of 3D graphics, low-level rendering, and C programming. The challenges helped us greatly, both as problem solvers and programmers. In the subsection below, we list the key challenges that we faced and how we were able to overcome them:

Common Errors & Solutions

1. Function Argument Mismatch – render_wireframe Update

Problem:

Once we added lighting support, we changed the render_wireframe() function to take a vec3_t light_pos parameter. This produced compilation errors such as "too many arguments" or "too few arguments" in demo files still using the previous function signature.

Solution:

We went through and updated all calls to render_wireframe() in all demo programs and verified the function declaration in renderer.h was identical to the new definition in renderer.c.

2. Implicit Declaration of Vector Math Functions

Problem:

These mistakes such as implicit function declaration of 'vec3_add' arose when using vector operations such as vec3_add, vec3_sub, and vec3_normalize in files such as renderer.c.

Solution:

We added correct declarations for all math3d helpers used for vector mathematics in math3d.h and made sure their implementations actually existed in math3d.c. Then, we added math3d.h in any file calling these functions.

3. Linking & Static Library Use

Problem:

Some demo programs didn't link or compile at all, especially when we attempted to compile them with a static library (libtiny3d.a).

Solution:

We conditioned our Makefile so there are specific rules for compiling the static library from individual object files. We also adjusted the demo's compilation process so it links against -Lbuild -ltiny3d for improved modularity and to link everything correctly.

4. Markdown Formatting Issues in README

Problem:

When writing the README.md, we initially exported it from Word, causing unnecessary formatting issues and incorrect file extensions that led to Markdown rendering failure on GitHub.

Solution:

We eschewed such plain text editors like Notepad and VS Code for file editing with the .md extension. This allowed for clean formatting and full compatibility with Markdown viewers and version control systems.

5. Animation Path Overlap

Problem:

In certain primitive animation tests, the cube and soccer ball would sometimes overlap or collide as they moved around the light source.

Solution:

We achieved this by assigning fixed angular offsets (i.e., 120 degrees apart) to the objects. This provided us with smooth, synchronized, and non-overlapping motion to all the animated objects.

Lessons Learned

- We learned a good grasp of 3D graphics fundamentals like transformations, projections, and lighting.
- We had direct exposure to modular C programming, header management, and static library linking.
- We learned the importance of tidy function interfaces and consistent documentation.
- We observed firsthand how versioning, tool selection (for example, code editor vs. word processor), and diligent debugging can quickly influence development process.

Overall, every challenge we worked on strengthened our understanding of programming graphics, and the exercise conditioned us to become more analytical in thought, clearer in team communication, and more systematic in solving actual coding issues.

11.Conclusion

The libtiny3d project has indeed been an eye-opener experience—both technically, and at a personal level. While the majority of programming projects are all about coding drill, this project needed us to build a 3D graphics engine from ground up in C without any use of any external high-level graphics libraries. This hands-on, in-depth exposure gave us so much better understanding of how computer graphics actually operate.

Through this process, we developed a firm understanding over some of the basics such as 3D vector and matrix computation, sub-pixel accurate line drawing, cubic Bézier animation, and the Lambertian lighting model. Concepts which at some stage appeared abstract in nature became something we could use confidently and in control.

Technically, we greatly enhanced in:

- Writing efficient and clean C code
- Creating low-level rendering pipelines
- Safe and efficient memory handling
- Mastering the mathematics of graphics systems

Other than the code, the project taught us that collaboration and communication are also essential. We were able to divide and conquer complicated tasks among ourselves, assisted one another to navigate bugs and design cycles, and collaborated to produce a polished final demo and detailed report.

At the end of the day, libtiny3d is more than a graphics engine—it's a reflection of how far we've progressed as students, coders, and problem solvers. This is a wonderful platform for future work in systems programming, real-time rendering, and computer graphics. We're pleased with what we've achieved, and even more so looking forward to what this will bring next.

Thank You!..