Department of Computer Engineering
Faculty of Engineering
University of Peradeniya
Sri Lanka

CO1020: Computer Systems Programming

C Project

Group 65

Team Members:

E/23/416 - Vishwaka A.G.S.

E/23/351 - Sanjuna K.D.

Contents

1 Introduction

This project **libtiny3d**, to build a lightweight 3D graphics engine from scratch in pure C– with no OpenGL, no DirectX, just math and pixels. The core goal is to deeply understand and implement the algorithms that transform mathematical descriptions of 3D objects into 2D images on a screen.

Instead of relying on established libraries, every component is hand-coded: a custom math library for vectors and matrices, a floating-point canvas for smooth line rendering, a complete software pipeline for 3D transformations and perspective projection, Lambertian lighting, and advanced animation using Bézier curves and quaternions. The final result is a functional demo showcasing animated, lit 3D wireframe objects, demonstrating the essential foundations of real-time graphics.

2 Implementation Details

2.1 Canvas and Line Drawing

The digital canvas and its associated line drawing functionality are designed to support smooth rendering even at sub-pixel precision. The core components include a canvas data structure, a function for setting individual pixel colors with bilinear filtering, and a line drawing algorithm utilizing the Digital Differential Analyzer (DDA) method.

2.1.1 `canvas_t` Structure Definition

The foundational element of this graphics system is the `canvas_t` structure, defined within `canvas.h`. This structure encapsulates the essential properties of the digital drawing surface:

```
typedef struct {
    unsigned char r, g, b;
} color_t;

typedef struct {
    int width;       // canvas width in pixels
    int height;      // canvas height in pixels
    color_t **pixels;    // 2d array for representing color at
       each pixel
} canvas_t;
```

- **width and height:** These integer members store the horizontal and vertical dimensions of the canvas, respectively, measured in pixels.
- **Pixels:** This member is a double pointer to `color_t` structures, effectively forming a 2D array. Each element `pixels[y][x]` represents a single pixel on the canvas and stores its color information. The `color_t` structure comprises three unsigned char components (r, g, b), each ranging from 0 to 255, allowing for a full spectrum of RGB color representation. This deviates from an initial specification that suggested floating-point brightness values, opting instead for a more comprehensive color model.

2.1.2 `set_pixel_f(canvas, x, y, color)` function

Located in `canvas.c`, the `set_pixel_f` function is responsible for accurately setting the color of a pixel at specified floating-point coordinates (x, y) while employing bilinear filtering to ensure smooth transitions.

```
ensure smooth transitions.
void set_pixel_f(canvas_t *canvas, float x, float y, color_t
   color) {
    // ... (implementation details)
}
```

Bilinear Filtering Mechanism

The primary goal of this mechanism is to distribute the input color across the four nearest integer pixels surrounding a floating-point coordinate (x, y), thereby reducing aliasing and producing smoother visual output.

- **Sub-pixel Precision:** Unlike standard pixel-aligned drawing, the `set_pixel_f` function accepts floating-point values for (x, y), allowing precise placement of colors between pixel centers.
- **Neighboring Pixel Identification:** The four integer pixel coordinates enclosing the floating-point target are determined using floor and ceil operations:

$$(\lfloor x \rfloor, \lfloor y \rfloor), \quad (\lceil x \rceil, \lfloor y \rfloor), \quad (\lfloor x \rfloor, \lceil y \rceil), \quad (\lceil x \rceil, \lceil y \rceil)$$

- **Weight Calculation:** The contribution (weight) of the input color to each surrounding pixel depends on the distance from (x, y) to that pixel's center.

Let,

$$dx = x - \lfloor x \rfloor, \quad dy = y - \lfloor y \rfloor$$

Then compute weights as,

$$\begin{aligned} w_{00} &= (1 - dx)(1 - dy) && \text{(for pixel } (\lfloor x \rfloor, \lfloor y \rfloor)) \\ w_{10} &= dx(1 - dy) && \text{(for pixel } (\lceil x \rceil, \lfloor y \rfloor)) \\ w_{01} &= (1 - dx)dy && \text{(for pixel } (\lfloor x \rfloor, \lceil y \rceil)) \\ w_{11} &= dx\, dy && \text{(for pixel } (\lceil x \rceil, \lceil y \rceil)) \end{aligned}$$

- **Color Distribution:** Each surrounding pixel's RGB components are incremented by the input color scaled by its corresponding weight. This effectively blends the color across pixels, creating a smooth transition.
- **Clamping:** To maintain valid color values within the 0–255 range, helper functions like `clamp_uchar` are used to prevent overflow or underflow during color accumulation.
- **Circular Viewport Constraint:** The process also incorporates a check using `is_pixel_in_circular_viewport` to ensure that only pixels within a circular region centered on the canvas are updated. This adds a deliberate visual constraint or stylistic effect to the rendering.

2.1.3 `draw_line_f(canvas, x0, y0, x1, y1, thickness, color)` function

Also implemented in `canvas.c`, the `draw_line_f` function renders a smooth line segment from a starting point (x_0, y_0) to an ending point (x_1, y_1) on the canvas, using a specified thickness and color. The Digital Differential Analyzer (DDA) algorithm is employed for line rasterization.

```
void draw_line_f(canvas_t *canvas, float x0, float y0,
                 float x1, float y1, float thickness, color_t
                     color) {
    // ... (implementation details)
}
```

DDA Algorithm Application

The DDA algorithm facilitates incremental line drawing by calculating pixel positions along the line.

- **Step Determination:** The algorithm first calculates the absolute differences in the x and y coordinates:

  $$dx = |x_1 - x_0|, \quad dy = |y_1 - y_0|$$

  The number of steps required is determined by:

  $$steps = \max(dx, dy)$$

  ensuring that at least one pixel is illuminated for each step along the dominant axis.

- **Incremental Values:** Incremental changes for x and y are computed by dividing the total displacement by the number of steps:

  $$x_{\text{inc}} = \frac{x_1 - x_0}{\text{steps}}, \quad y_{\text{inc}} = \frac{y_1 - y_0}{\text{steps}}$$

- **Iterative Pixel Drawing:** The function then iterates for the calculated number of steps. In each iteration, the current x and y coordinates are updated incrementally, and the `set_pixel_f` function is invoked with these updated coordinates and the specified color. This integration of `set_pixel_f` ensures that each "point" along the line benefits from bilinear filtering, contributing to the line's overall smoothness.

- **Thickness Implementation Note:** It is important to note that the current implementation of `draw_line_f` provides a simplified approach to line thickness. As indicated in the source code comments, it primarily renders a 1-pixel wide line by repeatedly calling `set_pixel_f`. Full support for variable line thickness, which would involve drawing wider lines or anti-aliased fills, would necessitate a more advanced line rendering algorithm.

2.1.4 Additional Utility Functions

The implementation also includes several auxiliary functions crucial for canvas management:

- `canvas_create(width, height)`: Dynamically allocates memory for a new `canvas_t` structure and its associated pixels array, initializing all pixel colors to black (RGB 0,0,0).
- `canvas_destroy(canvas)`: Deallocates all memory previously assigned to a `canvas_t` instance, preventing memory leaks.
- `canvas_clear(canvas)`: Resets the entire canvas by setting all pixel colors back to black.
- `canvas_save_pgm(canvas, filename)`: Facilitates the persistence of the canvas content by saving it to a PGM (Portable Graymap) file. During this process, the RGB `color_t` values are converted to grayscale by averaging their red, green, and blue components.

In conclusion, this code provides a robust framework for a digital drawing application, emphasizing smooth line rendering through bilinear filtering. While the line thickness feature is currently a simplification, the existing architecture offers a strong foundation for further graphical enhancements.

2.2 3D Math Foundation Details

This implementation provides a custom math engine to handle 3D vectors and 4x4 matrices, enabling fundamental transformations like translation, rotation, and scaling of 3D objects, along with perspective projection.

2.3 `vec3_t` structure (in `math3d.h`)

The `vec3` structure is designed to hold a 3D vector, supporting both Cartesian and spherical coordinate systems.

```
typedef struct {
    float x, y, z;         // Cartesian coordinates
    float r, theta, phi;   // Spherical coordinates
    bool cartesian_valid, spherical_valid; // Validity flags
} vec3;
```

- **x, y, z**: Cartesian coordinates.

- **r, theta, phi**: Spherical coordinates.
  - r: Radius (distance from origin).
  - θ: Polar angle (angle from the positive Z-axis).
  - ϕ: Azimuthal angle (angle from the positive X-axis in the XY-plane).
- **cartesian_valid, spherical_valid**: Flags used to track which coordinate representation is up-to-date. This ensures when one set is modified, the other is marked invalid and updated on access.

2.3.1 Coordinate Synchronization (in `math3d.c`)

The design ensures that when one coordinate system (Cartesian or spherical) is updated, the other is automatically synchronized upon request.

- `update_spherical_from_cartesian(vec3 *v)` (static helper):
  - Calculates r, θ, and ϕ from (x, y, z).
  - $$r = \sqrt{x^2 + y^2 + z^2}$$
  - $$\theta = \begin{cases} \arccos\left(\frac{z}{r}\right) & \text{if } r \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
  - $$\phi = \begin{cases} \text{atan2}(y, x) & \text{if } r \neq 0 \\ 0 & \text{otherwise} \end{cases}$$
  - Sets `v->spherical_valid = true`.
- `update_cartesian_from_spherical(vec3 *v)` (static helper):
  - Calculates x, y, and z from (r, θ, ϕ).
  - $$x = r\sin\theta\cos\phi, \quad y = r\sin\theta\sin\phi, \quad z = r\cos\theta$$
  - Sets `v->cartesian_valid = true`.
- `vec3_update_spherical(vec3 *v)`: Public function that calls `update_spherical_from_cartesian` if spherical coordinates are not valid.
- `vec3_update_cartesian(vec3 *v)`: Public function that calls `update_cartesian_from_spherical` if cartesian coordinates are not valid.

2.3.2 Vector Constructors (in `math3d.c`)

- `vec3_from_cartesian(float x, float y, float z)`: Creates a `vec3` from Cartesian coordinates and calls `vec3_update_spherical`.
- `vec3_from_spherical(float r, float theta, float phi)`: Creates a `vec3` from spherical coordinates and calls `vec3_update_cartesian`.
- `vec4_from_vec3(vec3 v, float w)`: Converts a `vec3` to a `vec4` by adding a w component for homogeneous transformations.

2.3.3 Vector Setters (in `math3d.c`)

- `vec3_set_cartesian(vec3 *v, float x, float y, float z)`: Sets Cartesian coordinates and invalidates spherical.
- `vec3_set_spherical(vec3 *v, float r, float theta, float phi)`: Sets spherical coordinates and invalidates Cartesian.

2.3.4 Vector Operations (in `math3d.c`)

These functions perform vector arithmetic, ensuring coordinate synchronization before operations.

- `vec3_add(vec3 a, vec3 b)`, `vec3_sub(vec3 a, vec3 b)`: Vector addition and subtraction.
- `vec3_dot(vec3 a,vec3 b)`, `vec3_cross(vec3 a,vec3 b)`: Dot and cross products.
- `vec3_length(vec3 v)`: Computes $\|v\|$.
- `vec3_normalize_fast(vec3 v)`, `vec3_normalize(vec3 v)`: Normalizes the vector, avoiding division by zero via epsilon check.
- `vec3_scale(vec3 v, float s)`: Scales by scalar.
- `vec3_slerp(vec3 a, vec3 b, float t)`: Spherical linear interpolation.
  - Smoothly interpolates between vectors a and b by fraction t.
  - Computes angle via arccos of dot product.
  - Projects and combines to yield interpolated vector.
- `vec3_from_vec4(vec4 v)`: Extracts a `vec3` from a `vec4`.

2.3.5 `mat4_t` Structure and Operations (in `math3d.h` and `math3d.c`)

The `mat4` structure represents a 4×4 matrix, essential for 3D transformations.

```
typedef struct {
    float m[16]; // Column-major order
} mat4;
```

Column-Major Order

Internally, the matrix elements are stored in a 1D array in column-major order. This means the elements are arranged in memory as:

$$\begin{bmatrix} m[0] & m[4] & m[8] & m[12] \\ m[1] & m[5] & m[9] & m[13] \\ m[2] & m[6] & m[10] & m[14] \\ m[3] & m[7] & m[11] & m[15] \end{bmatrix}$$

This layout aligns with conventions used by graphics APIs and simplifies matrix operations in transformation pipelines.

2.3.6 Matrix Operations

- `mat4_identity(void)`: Returns the identity matrix. An identity matrix has $1.0f$ on its main diagonal and $0.0f$ elsewhere. It represents no Transformation.
- `mat4_translate(float tx, float ty, float tz)`: Translation matrix, with tx, ty, tz in the last column.
- `mat4_scale(float sx, float sy, float sz)`: Scaling matrix with factors on the diagonal.
- `mat4_rotate_x(float angle)`, `mat4_rotate_y(float angle)`, `mat4_rotate_z(float angle)`: Rotation matrices using cos and sin of the angle.
- `mat4_rotate_xyz(float rx, float ry, float rz)`: Combines rotations in Z, Y, X order.
- `mat4_perspective(float l, float r, float b, float t, float n, float f)`: Constructs a perspective frustum matrix. This projects 3D points into 2D, using

  $$m[11] = -1, \quad m[14] = dependsonnear/far$$

  for perspective division.
- `mat4_mul(mat4 a, mat4 b)`: Multiplies two matrices.

- `mat4_mul_vec4(mat4 m, vec4 v)`: Multiplies a matrix by a 4D vector to apply transformations.

This math engine provides a comprehensive set of tools for 3D graphics operations, laying the groundwork for rendering and manipulating objects in a 3D environment.

2.4 3D Rendering Pipeline

This report outlines the implementation of a 3D rendering pipeline designed to transform and draw 3D objects as wireframes onto a 2D canvas. Key features include vertex projection, circular clipping, depth-sorted line rendering, and an optional quaternion-based rotation system.

2.4.1 `project_vertex()` function (in `renderer.c`)

The `project_vertex()` function is fundamental, transforming a 3D vertex from its local object space to clip space (homogeneous coordinates) through a series of matrix multiplications.

```
vec4 project_vertex(vec3 vertex, mat4 model_matrix, mat4
   view_matrix, mat4 projection_matrix) {
    // 1. Local to World (Model Transformation)
    vec4 v_local_homogeneous = vec4_from_vec3(vertex, 1.0f);
    vec4 v_world_homogeneous = mat4_mul_vec4(model_matrix,
       v_local_homogeneous);

    // 2. World to Camera (View Transformation)
    vec4 v_camera_homogeneous = mat4_mul_vec4(view_matrix,
       v_world_homogeneous);

    // 3. Camera to Clip (Projection Transformation)
    vec4 v_clip_homogeneous = mat4_mul_vec4(projection_matrix,
       v_camera_homogeneous);

    return v_clip_homogeneous;
}
```

Implementation Details:

- This function takes a `vec3` vertex and three `mat4` transformation matrices:
  - `model_matrix` (local to world),

  - `view_matrix` (world to camera),
  - `projection_matrix` (camera to clip).

- The vertex is first converted to a `vec4` (homogeneous coordinates, $w = 1.0$), then successively multiplied by each matrix.
- The final output is in clip space, ready for perspective division and viewport mapping to screen coordinates.

Circular Clipping via `clip_to_circular_viewport()`

Circular clipping is achieved by a helper function integrated within `canvas.c`.

```
// From canvas.c
static bool is_pixel_in_circular_viewport(canvas_t *canvas, int
   px, int py) {
    if (canvas == NULL) return false;
    float center_x = canvas->width / 2.0f;
    float center_y = canvas->height / 2.0f;
    float radius = fminf(center_x, center_y);
    float dist_sq = (px - center_x) * (px - center_x)
                  + (py - center_y) * (py - center_y);
    return dist_sq <= (radius * radius);
}
```

Implementation Details:

- Checks if a pixel (px, py) lies within a circular region centered on the canvas.
- All drawing functions (via `set_pixel_f`) internally use this, so lines are automatically clipped to a circular frame.

2.4.2 `render_wireframe()` function (in `renderer.c`)

The `render_wireframe()` function draws a 3D object as a wireframe on the 2D canvas, incorporating depth sorting and basic lighting.

```
void render_wireframe(canvas_t *canvas, object3d_t *object,
                      mat4 model_matrix, mat4 view_matrix,
                      mat4 projection_matrix, float
                         line_thickness,
                      vec3* light_dirs, int num_lights) {
    // ... (implementation details)
}
```

Implementation Details:

- **Vertex Projection:**
  - All vertices are transformed to clip space using `project_vertex()`.
- **Culling and Transformation:**
  - Edges are iterated, frustum culling skips lines entirely outside view or behind near plane.
  - Valid clip coordinates are converted to Normalized Device Coordinates (NDC) by dividing by w:

$$x_{ndc} = \frac{x_{clip}}{w_{clip}}, \quad y_{ndc} = \frac{y_{clip}}{w_{clip}}, \quad z_{ndc} = \frac{z_{clip}}{w_{clip}}$$

  - Then mapped to screen coordinates.
- **Lighting:**
  - Intensity is calculated per edge based on dot products with light directions, boosting visibility with a power function.
- **Depth Sorting:**
  - Each line stores an average z_{ndc}.
  - Lines are sorted back-to-front using `qsort()`.
- **Drawing:**
  - Sorted lines are drawn with `draw_line_f()` with their computed thickness and color.

2.4.3 `generate_soccer_ball()` function (in `main.c`)

This function creates the geometric data for the 3D object.

```
object3d_t* generate_soccer_ball() {
    // ... (implementation details)
}
```

Implementation Details:

- A true soccer ball would be a truncated icosahedron (60 vertices, 90 edges).
- For simplicity, this implementation generates a cube (8 vertices, 12 edges) to demonstrate the rendering pipeline.

2.4.4 Quaternion Rotation

Quaternion rotation is implemented to provide smooth, gimbal-lock-free rotation.

- The `quat_t` structure stores (x, y, z, w).
- Functions include:
  - `quat_identity()`, `quat_from_axis_angle()`
  - `quat_mul()` - combines rotations
  - `quat_normalize()`
  - `quat_to_mat4()` - convert to rotation matrix
  - `quat_slerp()` - for smooth interpolation
- Used in `main.c` to update rotation each frame and convert to a model matrix.

2.4.5 Demo (`main.c`)

The `main.c` file runs the rendering demo of a rotating wireframe clipped to a circle.

Flow:

- Initializes canvas, generates the 3D object.
- Sets up camera with `mat4_look_at` and projection with `mat4_perspective_fov`.
- Defines a directional light.
- In each animation frame:
  - Updates rotation with quaternions.
  - Clears canvas.
  - Calls `render_wireframe()`.
  - Saves canvas to a PGM file.
- PGM files can be combined externally into an animation, demonstrating the full 3D rendering pipeline.

2.5 Lighting & Polish

This section details the implementation of Lambertian lighting for wireframes and cubic Bézier curve animation, designed to enhance the visual quality and smoothness of 3D object rendering.

2.5.1 Lambert Lighting (in `lighting.c` and `lighting.h`)

The `lighting.c` file contains the `compute_edge_lighting` function, which calculates the brightness of each wireframe edge based on its orientation relative to light sources.

Header Declaration (`lighting.h`)

```
// ... (snippet)
float compute_edge_lighting(vec3 v1, vec3 v2, vec3* light_dirs,
   int num_lights);
// ...
```

Implementation (`lighting.c`)

```
float compute_edge_lighting(vec3 v1, vec3 v2, vec3* light_dirs,
   int num_lights) {
    vec3 edge_dir = vec3_normalize_fast(vec3_sub(v2, v1));
    float intensity = 0.0f;
    for (int i = 0; i < num_lights; ++i) {
        float dot_val = vec3_dot(edge_dir, light_dirs[i]);
        intensity += fmaxf(0.0f, dot_val);
    }
    if (num_lights > 0) {
        intensity /= (float)num_lights;
    }
    const float ambient = 0.2f;
    return fminf(1.0f, ambient + intensity * (1.0f - ambient));
}
```

Implementation Details:

- **Edge Direction as "Normal":** Approximates an edge's normal using its normalized direction vector.
- **Lambertian Model:** Calculates the dot product between the edge direction and each light direction. Only positive contributions (facing the light) add to intensity, achieving a Lambertian effect.

- **Multiple Lights:** Intensity from all lights is summed and normalized by the number of lights to avoid over-brightness.
- **Ambient Lighting:** Adds a base ambient term (0.2) and clamps the result to [0,1] to ensure valid brightness.

2.5.2 Bézier Animation (in `animation.c` and `animation.h`)

The `animation.c` file implements the `bezier_cubic` function, which calculates a point along a cubic Bézier curve, enabling smooth object movement over time.

Header Declaration (`animation.h`)

```
// ... (snippet)
vec3 bezier_cubic(vec3 p0, vec3 p1, vec3 p2, vec3 p3, float t);
// ...
```

Implementation (`animation.c`)

```
vec3 bezier_cubic(vec3 p0, vec3 p1, vec3 p2, vec3 p3, float t) {
    t = fmaxf(0.0f, fminf(1.0f, t)); // Clamp t to [0, 1]
    float one_minus_t = 1.0f - t;
    float one_minus_t_sq = one_minus_t * one_minus_t;
    float t_sq = t * t;

    // Bernstein polynomials
    float c0 = one_minus_t_sq * one_minus_t;    // (1-t)^3
    float c1 = 3.0f * one_minus_t_sq * t;       // 3*(1-t)^2*t
    float c2 = 3.0f * one_minus_t * t_sq;       // 3*(1-t)*t^2
    float c3 = t_sq * t;                        // t^3

    // Compute interpolated point
    float x = c0 * p0.x + c1 * p1.x + c2 * p2.x + c3 * p3.x;
    float y = c0 * p0.y + c1 * p1.y + c2 * p2.y + c3 * p3.y;
    float z = c0 * p0.z + c1 * p1.z + c2 * p2.z + c3 * p3.z;

    return vec3_from_cartesian(x, y, z);
}
```

Implementation Details:

- **Cubic Bézier Formula:**

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3$$

- **Clamping:** Ensures t stays within [0,1].
- **Blending:** Uses Bernstein polynomials to compute the contribution of each control point.
- **Smooth Paths:** Generates smooth positions ideal for continuous animation.

2.5.3 Integration and Demo (`main.c`)

Lambert Lighting Application:

- In `render_wireframe`, the `compute_edge_lighting` function is called for each edge.
- The resulting intensity sets the grayscale color, making lines brighter when facing light sources.

Bézier Animation for Multiple Objects:

- Control points (p_0, p_1, p_2, p_3) define animation paths.
- A normalized time t is calculated per frame.
- `bezier_cubic()` computes the object's position on the curve.
- A translation matrix from this position combines with the rotation matrix (e.g., from quaternions) to form the final `model_matrix`.
- Each object is then rendered, resulting in smooth, synchronized motion.

These implementations collectively enhance the visual appeal of the wireframe renderer by adding realistic lighting effects and enabling sophisticated, smooth animations of 3D objects.

3 Mathematical Background for 3D Graphics

This section delves into the mathematical foundations underpinning the 3D rendering pipeline, specifically focusing on 3D vectors, 4x4 matrices, quaternions, and Bézier curves. These concepts are essential for representing objects in 3D space, transforming them, and animating their movement.

3.1 Vectors and Their Operations

Vectors are fundamental to 3D graphics, used to represent positions, directions, and forces.

3.1.1 vec3 and vec4 Structures

- **vec3:** Represents a 3D vector. It stores coordinates in two forms:
  - **Cartesian Coordinates** (x, y, z): Defines a point's position along three perpendicular axes.
  - **Spherical Coordinates** (r, θ, ϕ):
    - r: Radial distance from the origin.
    - θ: Polar angle (angle from the positive Z-axis), $0 \leq \theta \leq \pi$.
    - ϕ: Azimuthal angle (angle from the positive X-axis in the XY-plane), $0 \leq \phi < 2\pi$.

  The implementation uses flags (`cartesian_valid`, `spherical_valid`) to keep these representations synchronized.

- **vec4:** Represents a 4D vector for homogeneous coordinates:
  - For points, $w = 1$.
  - For directions, $w = 0$.

  Homogeneous coordinates allow all affine transformations to be represented as matrix multiplications. After transformations, a perspective divide converts back to 3D.

3.1.2 Vector Operations

- **Addition and Subtraction:**

$$A + B = (A_x + B_x, A_y + B_y, A_z + B_z)$$

$$A - B = (A_x - B_x, A_y - B_y, A_z - B_z)$$

- **Scalar Multiplication:**

$$s \cdot V = (sV_x, sV_y, sV_z)$$

- **Dot Product:**

$$A \cdot B = A_x B_x + A_y B_y + A_z B_z = |A||B| \cos \alpha$$

  Useful for computing angles, projections, and lighting calculations.

- **Cross Product:**

$$A \times B = (A_y B_z - A_z B_y, A_z B_x - A_x B_z, A_x B_y - A_y B_x)$$

  Its magnitude equals $|A||B| \sin \alpha$ and is used for normals and orthogonal vectors.

- **Length (Magnitude):**

$$|V| = \sqrt{V_x^2 + V_y^2 + V_z^2}$$

- **Normalization:**

$$\hat{V} = \frac{V}{|V|}$$

  Produces a unit vector.

- **Spherical Linear Interpolation (SLERP):** For two unit vectors A and B:

$$SLERP(A, B, t) = \frac{\sin((1-t)\theta)}{\sin \theta} A + \frac{\sin(t\theta)}{\sin \theta} B$$

  where $\theta = \arccos(A \cdot B)$, ensuring smooth constant-speed interpolation on the sphere.

3.2 Matrices and Quaternions

3.2.1 mat4 Structure and Operations

A `mat4` represents a 4×4 matrix in column-major order, essential for transformations.

- **Identity Matrix:** Leaves vectors unchanged.
- **Translation Matrix:** Shifts by (t_x, t_y, t_z) with translation components placed in the last column.
- **Scaling Matrix:** Scales by (s_x, s_y, s_z) with scaling factors on the diagonal.
- **Rotation Matrices:** Rotate around X, Y, or Z axes using sin and cos.
- **Combining Rotations:**

  $$\texttt{mat4\_rotate\_xyz}(r_x, r_y, r_z)$$

  multiplies rotation matrices, where order matters due to non-commutativity.
- **Matrix Multiplication:**

  $$C = A \times B$$

  composes transformations, applying B first, then A.
- **Matrix-Vector Multiplication:**

  $$V' = M \times V$$

  applies transformations to vectors.

3.2.2 Projection Matrix

The function

$$\texttt{mat4\_perspective}(l, r, b, t, n, f)$$

builds a perspective projection matrix. It maps the frustum defined by:

- l, r: left and right clipping planes
- b, t: bottom and top clipping planes
- n, f: near and far clipping planes

into canonical view space, simulating perspective.

3.2.3 Quaternions

Quaternions (x, y, z, w) elegantly represent rotations.

- Avoid gimbal lock common in Euler angles.
- Support smooth interpolation (SLERP) over the shortest path on the 4D hypersphere.
- Operations include:
  - Identity quaternion $(0, 0, 0, 1)$
  - Conversion from axis-angle
  - Quaternion multiplication to combine rotations
  - Normalization to maintain unit length
  - Conversion to rotation matrix for integration with `mat4`

3.3 Bézier Curves

Bézier curves provide smooth interpolation paths.

3.3.1 Cubic Bézier Formula

Given four control points P_0, P_1, P_2, P_3, the curve is:

$$B(t) = (1-t)^3 P_0 + 3(1-t)^2 t P_1 + 3(1-t)t^2 P_2 + t^3 P_3 \quad for t \in [0, 1]$$

where

- $(1-t)^3$, $3(1-t)^2 t$, $3(1-t)t^2$, and t^3 are Bernstein polynomials acting as blending weights.
- As t varies from 0 to 1, the curve starts at P_0 and smoothly transitions to P_3.

3.3.2 Smooth Animation

By incrementally varying t over time, we generate natural motion paths for objects or cameras, enabling fluid animations that enhance visual appeal.

These mathematical components collectively form the backbone of the 3D rendering pipeline, enabling accurate representation, transformation, and dynamic animation of objects in virtual 3D environments.

4 Design

The design of this graphics engine (libtiny3d) was carefully structured to build a complete rendering pipeline purely in C, without relying on graphics APIs like OpenGL or DirectX. This required a bottom-up approach, starting from basic mathematical representations up to high-level rendering logic.

4.1 Modular Separation

The engine was divided into several distinct modules, each addressing a specific aspect of the 3D graphics pipeline:

canvas Implements the digital drawing surface as a 2D pixel buffer. It provides functions for setting pixels with bilinear filtering (for sub-pixel precision), clearing the canvas, and exporting the image (e.g., to PGM files). This is the final stage where all computed colors are written.

4.1.1 math3d

Serves as the core computational library. It defines:

- vec3 and vec4 structures for representing 3D and 4D points/vectors.
- mat4 for 4×4 matrices used in transformations (translation, rotation, scaling, projection).
- quat for quaternions, enabling robust rotation without gimbal lock.

4.1.2 renderer

Responsible for projecting 3D geometry onto the 2D canvas. It performs:

- Transformations from model space to world, camera, and clip space via matrix multiplication.
- Perspective division and viewport mapping to convert 3D positions to 2D screen coordinates.
- Line drawing using the DDA algorithm combined with sub-pixel bilinear filtering to ensure smooth, anti-aliased lines.

4.1.3 lighting

Implements a simplified Lambertian model. It computes intensity for each edge by taking the dot product of the edge's direction and light directions, combined with ambient light. This gives the wireframe a shaded look, improving depth perception.

4.1.4 animation

Provides Bézier curve interpolation to animate translations smoothly along a path, and quaternion-based rotation to animate orientations without sudden jumps or gimbal lock. This allows multiple objects to follow different trajectories in a synchronized manner.

5 Challenges and Solutions

5.1 Vector Operations and Synchronization

Challenge:

Keeping Cartesian and spherical coordinate representations consistent within the vector data structure demanded precise conversion functions and control over when each representation is updated. Inconsistent updates could lead to transformation errors or lighting artifacts.

Solution:

Introduced validity flags and static helper functions to update spherical coordinates only after Cartesian changes and vice versa. This lazy update mechanism avoids unnecessary recomputations while ensuring both coordinate systems stay synchronized accurately.

5.2 Understanding Matrix Transformations for Movement, Rotation, and Lighting

Challenge:

Comprehending how model, view, and projection matrices interact to produce correct transformations was difficult. Correct order of matrix multiplication and integrating lighting computations in the proper coordinate space required deep understanding.

Solution:

Decomposed matrix operations into isolated, testable functions and visualized intermediate results to validate correctness. Used extensive external references such as Wikipedia articles, online tutorials, and educational videos to build intuition and confirm implementation accuracy.

5.3 Generating Ball Shape

Challenge:

Creating the geometric data for a soccer ball, which is a truncated icosahedron with 60 vertices and 90 edges, involves complex mathematical modeling and precise edge definitions. This complexity makes direct coding or manual data entry impractical and error-prone.

Solution:

Initially, extensive research was done through YouTube tutorials, Wikipedia articles, and web resources to understand the mathematical properties of icosahedrons and truncated icosahedrons. First, an icosahedron was programmatically generated as the base polyhedron. Using this, the truncated icosahedron structure of the soccer ball was constructed by applying vertex truncation and edge definitions, enabling an accurate representation of the ball's geometry in the renderer.

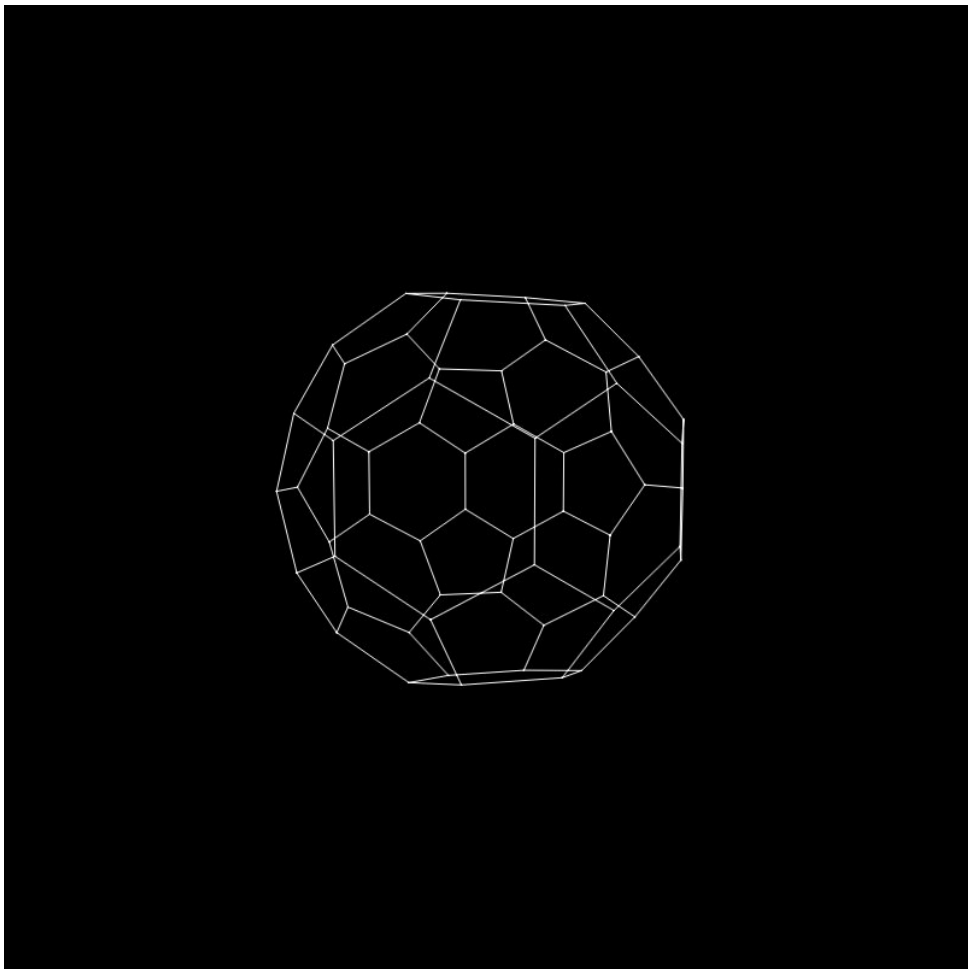

Figure 1: A rendered soccer ball wireframe

5.4 Animation Loop Management

Challenge:

Implementing a continuous, smooth animation loop that updates object rotation each frame while maintaining synchronization with rendering and frame saving required careful handling of quaternion operations and model matrix updates.

Solution:

Developed a frame-based loop where the rotation quaternion is updated incrementally and normalized each iteration. This quaternion is then converted to a rotation matrix and combined with translation matrices to update the model matrix. Rendered frames are saved as PGM images for external animation creation.

5.5 Working Across Multiple Folders

Challenge:

Organizing source code, headers, demos, and tests into separate folders introduced complexity in build automation, dependency tracking, and include path management. Incorrect configurations caused build failures or linker errors.

Solution:

Defined a clear folder structure with dedicated Makefiles that specify correct include paths and dependencies. This modular approach enabled seamless compilation and linking from the project root, improving build speed, clarity, and maintainability.

5.6 Visualizing the Canvas Output

Challenge:

Since the rendering output was in the PGM image format, which is not widely supported by common image viewers, verifying and reviewing rendered frames became cumbersome.

Solution:

Created Python scripts to convert sequences of PGM frames into animated GIFs, enabling easy animation previews. Additionally, leveraged online converters to transform PGM files into PNG images for easier visualization and sharing.

6 AI Tool Usage

Throughout this project, we extensively utilized AI tools (ChatGPT and Gemini) to support both the learning and implementation phases. This assistance covered a wide range of technical and documentation-related areas, enabling us to complete the project more efficiently and with deeper understanding. Key areas include:

- **Understanding 3D Graphics Fundamentals:** AI tools provided explanations and tailored examples on how 3D rendering pipelines function, the role of vertex transformations, clip space, and how wireframe representations are generated, building our foundational knowledge.
- **Matrix Transformations for Movement and Rotation:** When learning how to correctly apply and combine model, view, and projection matrices, AI gave step-by-step guidance on constructing translation, rotation, and scaling matrices, as well as how to apply them in the right order to achieve expected transformations.
- **Advanced Matrix Mathematics:** We explored homogeneous coordinates, perspective projection, and how matrices facilitate lighting calculations. AI also clarified subtle points such as why we perform the perspective divide and how to maintain proper lighting direction transformations across different spaces.
- **Code Debugging and Optimization:** AI tools helped us reason through compiler errors, runtime segmentation faults, and even suggested optimizations for vector and matrix operations to ensure both correctness and performance.
- **Designing Smooth Looping Animations:** To achieve continuous, visually appealing animations, especially with quaternion-based rotations and Bézier interpolations, AI assisted in fine-tuning the mathematical formulas and iteration schemes.
- **Creating the Soccer Ball Geometry:** AI tools were extensively used to streamline the process of integrating the soccer ball model. After generating the initial icosahedron programmatically, we decided to load a truncated icosahedron from an OBJ file to achieve the soccer ball structure. Here, AI helped by generating code snippets to parse the OBJ file, extract vertex data, and automatically build the appropriate edge lists required by our renderer. Additionally, AI was used to suggest efficient ways to update the `main.c` to incorporate these dynamically generated edges, ensuring smooth integration into the rendering loop without disrupting existing transformations or animations.
- **Improving Rendering and Lighting Techniques:** The Lambertian lighting implementation, including support for multiple light sources and how to blend ambient

with diffuse components, was iteratively improved through examples and alternative approaches provided by AI.

- **Generating and Refining Code Snippets:** For common but intricate functions—such as vector normalization, dot and cross products, quaternion operations, and bilinear filtering for sub-pixel accuracy—AI assisted by writing, reviewing, and explaining small, reusable C functions.
- **Organizing the Report Structure:** ChatGPT was used to plan the logical progression of report sections, ensuring a smooth flow from introduction, implementation details, mathematical background, to challenges and conclusions.
- **LaTeX Report Writing Assistance:** Specific LaTeX help included formatting multi-level sections and lists, generating mathematical equations, and writing c codes in the report.
- **Project Setup and Build System:** AI tools were also leveraged to create a robust **Makefile** for compiling multiple source files into both object files and a static library (**libtiny3d.a**), streamlining the build process. Similarly, AI provided structured templates and phrasing for the **README.md** to document the project effectively, and assisted in designing **tiny3d.h** as a unified header to expose core functionalities of the graphics engine.

7 Individual Contributions

7.1 E/23/351 - Sanjuna K.D.

- Implemented Task 1 and Task 2 of the project, covering the `canvas` and `math3d` modules.
- Created the `README.md` and `Makefile` to document and automate the build process.
- Generated the icosahedron and extended it to construct the soccer ball (truncated icosahedron).
- Authored the *Design*, *Challenges and Solutions*, and *AI Tool Usage* sections of the report.
- Composed the report in LaTeX.
- Contributed to editing and producing portions of the project demonstration video.

7.2 E/23/416 - Vishwaka A.G.S.

- Implemented Task 3 and Task 4 of the project, including the `renderer`, `lighting`, and `animation` modules.
- Enhanced and refined parts of the `canvas` and `math3d` modules after initial development.
- Developed the cube example in the `tests` directory to validate the rendering pipeline.
- Authored the *Implementation Details* and *Mathematical Background for 3D Graphics* sections of the report.
- Contributed to editing and producing portions of the project demonstration video.

8 References

1. J. Schneide, *Generating an Icosphere in C*, 2016. Available: https://schneide.blog/2016/07/15/generating-an-icosphere-in-c/

2. D. Video, *Generating an Icosahedron*, YouTube, 2023. Available: https://youtu.be/vqBK_PfXAok?si=6-Ly9XQMAE49PhKx

3. StackOverflow, *Three.js MeshPhongMaterial wireframe lighting*, 2017. Available: https://stackoverflow.com/questions/43508382/three-js-meshphongmaterial\-wireframe-lighting

4. Polyhedra Tessera, *Truncated Icosahedron Operations*, Available: https://polyhedra.tessera.li/truncated-icosahedron/operations

5. D. Video, *How to create a Soccer Ball Geometry*, YouTube, 2023. Available: https://youtu.be/p09i_hoFdd0?si=XbQjDpE74tinKALD

6. M. Author et al., *Mathematical Foundations of Polyhedral Geometry*, HAL, 2014. Available: https://hal.science/hal-01092485/file/preprint2.pdf

7. VitaminAC, *Matrices in Computer Graphics*, 2023. Available: https://vitaminac.github.io/posts/Matrices-in-Computer-Graphics/

8. FinalMesh, *Understanding WebGL Matrices*, Available: https://www.finalmesh.com/docs/webgl/matrix.htm

9. GameMath, *Matrix Introduction in Computer Graphics*, Available: https://gamemath.com/book/matrixintro.html

10. Unity Technologies, *Universal Render Pipeline*, Available: https://unity.com/features/srp/universal-render-pipeline