Department of Computer Engineering
University of Peradeniya
C Project
CO1020 : Computer Systems Programming

Group 56
Weerasinghe W.P.T.H. – E/22/421
Kariyawasam K.P.W.D.R. – E/22/182

Task 1: Canvas & Line Drawing

Introduction

The foundation of any 2D or 3D rendering system is the canvas, which is a digital surface onto which graphical primitives are drawn. In this task, a flexible and high-precision canvas system was implemented, designed to support smooth line rendering with subpixel accuracy. Unlike traditional rasterization approaches that use integer coordinates and discrete pixels, this system supports floating-point precision and distributes brightness smoothly across neighboring pixels using bilinear filtering. This canvas system is used as the graphical output layer for the entire rendering library and provides the visual basis for rendering both static and animated 2D or 3D wireframe shapes.

Implementation Details

This section implements the core functionality of the canvas system used in the 3D rendering library. The system is designed to simulate a high-resolution digital surface where each pixel can hold a grayscale brightness value, allowing for smooth and anti-aliased line rendering using subpixel precision.

***canvas_t* Structure:**

This structure stores essential attributes of the canvas.

- Width and height: Dimensions of the canvas
- Pixels: In this we store a dynamically allocated 2D array of floats representing grayscale intensity values (0.0 to 1.0) for each pixel

Now let's look into the specific functions itself,

***create_canvas(width, height)*:**

This function initializes a new canvas structure by dynamically allocating memory for the 2D pixels array and sets all pixel values to black. Each row is allocated individually using calloc to ensure zero-initialization.

***clear_canvas(canvas)*:**

Releases the memory allocated for the canvas by freeing each row of the pixel array and finally freeing the main pixels array and the canvas_t structure itself.

***set_pixel_f(canvas, x, y, intensity)*:**

This function sets pixel intensity using bilinear filtering by following these steps,

- Accepts floating-point (*x, y*) coordinates.
- Distributes the brightness value across the four surrounding pixels proportionally using their fractional distances (*dx, dy*) from the actual floating-point coordinate.
- A small *smoothness* offset is added to ensure visible softness even for minimal intensities.

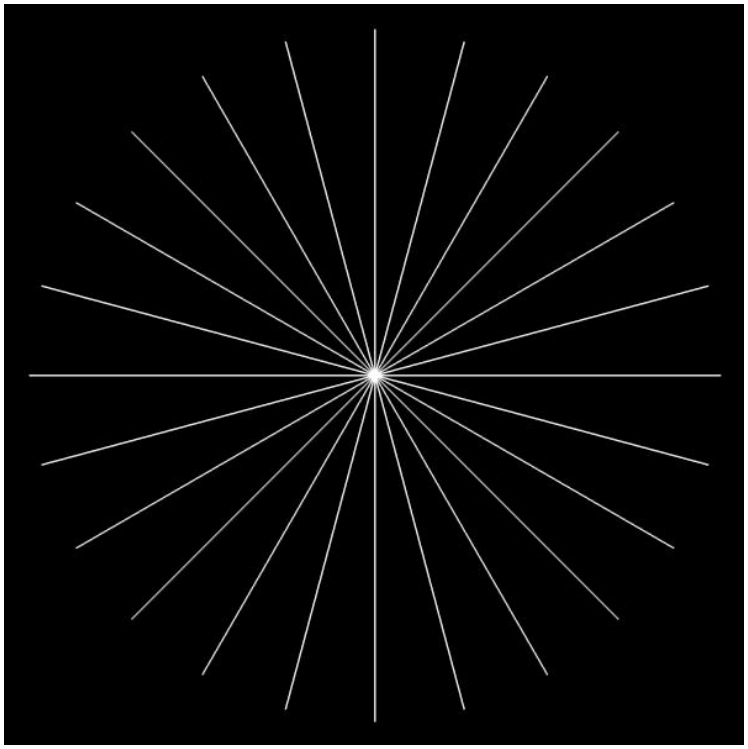
This technique produces smooth visual results compared to simply rounding to the nearest pixel.

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

This function draws smooth, thick lines using a variation of the DDA algorithm by calculating the number of steps required based on the larger of the x or y deltas and stepping along the line line incrementally, computing (*x, y*) for each point a square region of pixels (defined by the given thickness) is considered. Finally, each pixel within the circular region (determined by *dx^2 + dy^2 <= r^2*) is set using *set_pixel_f()*.

Together, these functions provide a robust and high quality rendering surface capable of drawing smooth geometric primitives, which serves as the foundation for the entire 3D rendering pipeline described in later tasks.

Demo:



Task 2: 3D Math Foundation

Introduction

A unique 3D math engine was created from the ground up to address the requirements of offering the necessary tools for managing 4x4 transformation matrices and 3D vectors. For operations like object movement, rotation, scaling, and camera projection, this engine provides the computational foundation.

The core of the vector component is the vec3 structure, which automatically synchronizes the two coordinate systems and supports both spherical (r, θ, ϕ) and Cartesian (x, y, z) representations. More adaptable operations, like simple animation and direction control, are made possible by this dual format capability.

Implementation Details

This section describes the detailed implementation of the 3D math engine, which supports both vector and matrix operations critical for transforming objects in a 3D space.

Vector Structure (*vec3*):

We defined a unique structure to support the definition of vectors in both the Cartesian and spherical coordinate systems. A union is used to overlay both representations, allowing seamless access to either form. In order to track the current active format, we implemented the *is_cartesian* flag which will be 1 if it's Cartesian and 0 if it's spherical.

***vec3_from_spherical(float r, float theta, float phi)*:**

This function converts spherical coordinates to Cartesian form using standard trigonometric transformations.

$$x = r * \sin(\phi) * \cos(\theta)$$
$$y = r * \sin(\phi) * \sin(\theta)$$
$$z = r * \cos(\phi)$$

***fast_inv_sqrt(float x)*:**

This function computes the fast inverse square root of a number. It approximates $1/\sqrt{x}$ efficiently and is used to normalize vectors quickly. The method involves treating a float as an integer to compute an initial guess, then refining it with one Newton-Raphson iteration.

***vec3_normalize_fast(vec3 v)*:**

This function normalizes the given vector *v* using the *fast_inv_sqrt()* method. It avoids the computational cost of *sqrt()* by leveraging the fast inverse square root, making it suitable for applications like real-time graphics.

***vec3_slerp(vec3 a, vec3 b, float t)*:**

This function performs Spherical Linear Interpolation (SLERP) between vectors *a* and *b*, based on a factor *t* in the range [0, 1]. This is ideal for smooth interpolation of directions or orientations.

- For nearly identical vectors (*dot > 0.9995*), simple linear interpolation followed by normalization is used.
- For general cases, the interpolation uses the spherical formula to ensure constant angular velocity, maintaining smoothness and realism in transitions.

Matrix Structure (*mat4*):

This structure was defined to encapsulate a 4×4 transformation matrix. In this structure, the matrix is stored in the column-major order. This supports composite transformations and perspective projections.

***mat4_identity()*:**

This function creates and returns the identity matrix. This is essential because the identity matrix is the neutral element for matrix multiplication, often used as the starting point for applying transformations.

***mat4_mul(mat4 a, mat4 b)*:**

This function multiplies two 4×4 matrices *a* and *b* and returns the result. This is implemented by summing the products of rows from *a* and columns from *b*, consistent with column-major layout. This operation is essential for combining transformations.

***mat4_translate(float tx, float ty, float tz)*:**

This function constructs a translation matrix that moves objects by *(tx, ty, tz)*. The translation components are inserted into the last column of the matrix, enabling vector movement in world or object space.

***mat4_scale(float sx, float sy, float sz)*:**

This function creates a scaling matrix that stretches or shrinks objects along the x, y, and z axes. The scale factors are placed along the diagonal of the matrix.

***mat4_rotate_xyz(float rx, float ry, float rz)*:**

This function generates a composite rotation matrix from Euler angles *rx, ry,* and *rz,* which represent rotations about the X, Y, and Z axes respectively. The method initially involves creating individual rotation matrices (*Rx, Ry, Rz*) for each axis and then multiplying them in the order of *Rz * Ry * Rx*.

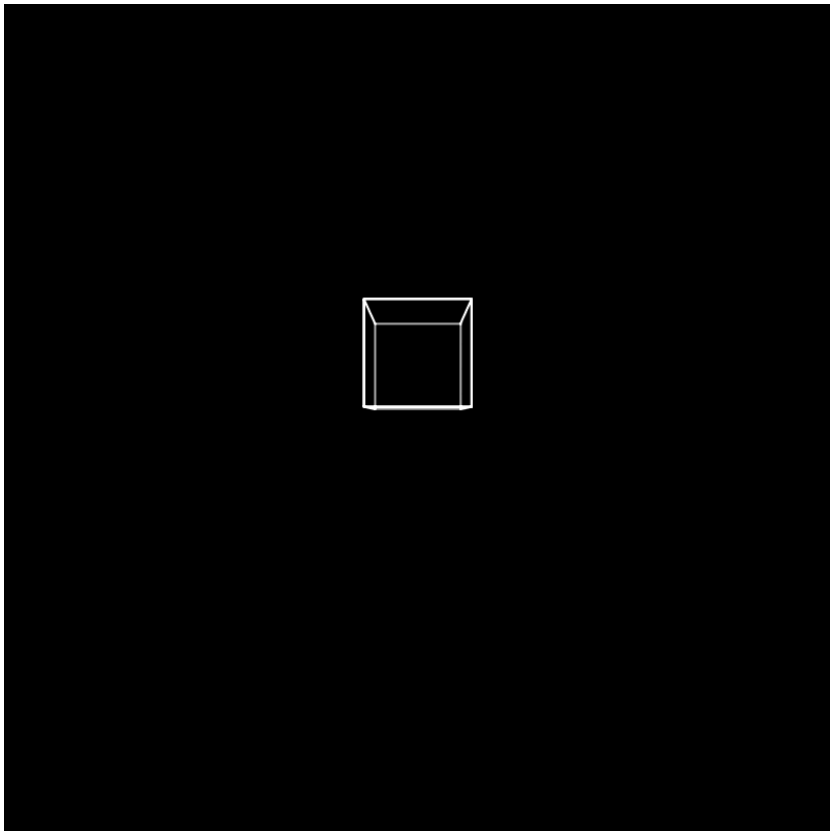
***mat4_frustum_asymmetric(float left, float right, float bottom, float top, float near, float far)*:**

This function creates a perspective projection matrix with an asymmetric frustum. This allows for off-center camera projections or special effects like stereo rendering. It is defined by near/far plane distances and screen bounds (*left, right, bottom, top*). The resulting matrix maps 3D points into clip space for rendering.

***mat4_apply_to_vec3(mat4 m, vec3 v)*:**

This function is where we apply all the transformations to the vector itself. It applies a 4×4 matrix transformation to a 3D vector *v*. This function includes perspective division which initially computes the homogeneous w-coordinate and then divides the transformed x, y, z components by w to return to Cartesian space. This is crucial when applying projection matrices, ensuring correct perspective scaling based on depth. A safety check is also included to prevent division by zero.

Demo:



Task 3: 3D Rendering Pipeline

Introduction

In this section we were required to create a 3D rendering pipeline responsible for converting a three-dimensional object into a two-dimensional visual representation on screen. In this project, a custom pipeline was implemented to transform, project, and draw 3D wireframe models onto a 2D canvas, while handling critical aspects such as spatial transformation, depth sorting, and viewport clipping.

Implementation details

This section describes how we implemented 3D to 2D transformations and Wireframe drawing to ultimately generate animated frames of a hardcoded geometrical object. This section could be considered as the core of the 3D rendering pipeline.

3D to 2D Transformation and Wireframe Drawing

The *renderer.c* file contains the core of the 3D rendering pipeline, projecting 3D geometry to 2D space and drawing it on the canvas.
Let's briefly go through some of the key functions utilized in this pipeline,

***project_vertex()*:**

This function applies the complete transformation pipeline for,
Local → World → Camera → Projection
The resulting homogeneous coordinates are mapped to screen coordinates. The Y-axis is flipped to match screen space convention.

***clip_to_circular_viewport()*:**

This function ensures that only pixels within a circular region at the center of the canvas are rendered. This in turn will mimic a circular lens effect.

***render_wireframe()*:**

This is one of the most crucial functions which draws each edge of the 3D model as a line on the 2D canvas using the following steps,

1. Transform Vertices:
    - *mat4_apply_to_vec3()* is used to apply world and local transforms to 3D points.
    - Each transformed vertex is then projected to 2D screen space.

2. Edge Sorting by Distance:
    - Each edge's midpoint is calculated and its distance squared from a reference point (the camera or light source) is computed.
    - Edges are sorted in descending order of depth to enable painter's algorithm rendering (back-to-front).

3. Drawing with Depth-based Intensity:
    - For every few edges, the intensity is recalculated based on distance using *map_value()*
    - Lines are rendered using *draw_line_f()* only if both endpoints are within the circular viewport.

***map_value()*:**

This is a utility to remap a value from one range to another. This is used to convert depth values into appropriate brightness intensities.

Application Entry and Frame Generation

The *main.c* file serves as the entry point of the rendering system, handling scene setup, animation over time, and outputting a sequence of frames as PPM images for later compilation into an animation.

Let's look into the key components of this file,

Screen Configuration:

```
int SCREEN_WIDTH = 800;
int SCREEN_HEIGHT = 800;
```

This sets the resolution of the canvas to render on. In all of our renders, we opted to keep it at a constant 800 pixels.

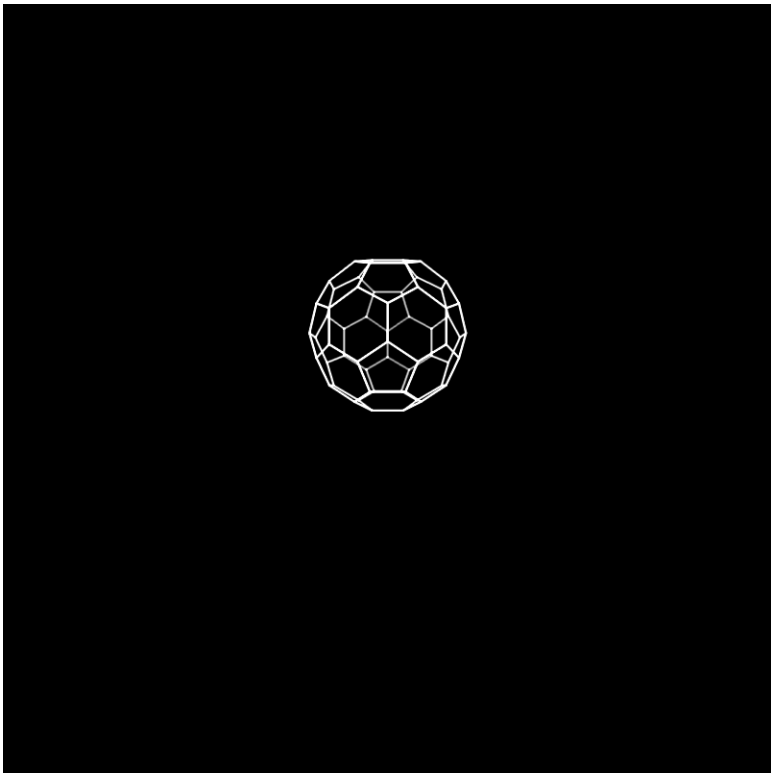
Vertex and Edge Data:

We chose to hardcode the geometrical object we want to animate. We can do that by defining the vertices and edges of the object and by setting the parameters accordingly. In our soccer ball implementation, we defined all the 60 vertices and 90 edges of a truncated icosahedron. Each vertex uses a *vec3* structure with Cartesian coordinates.

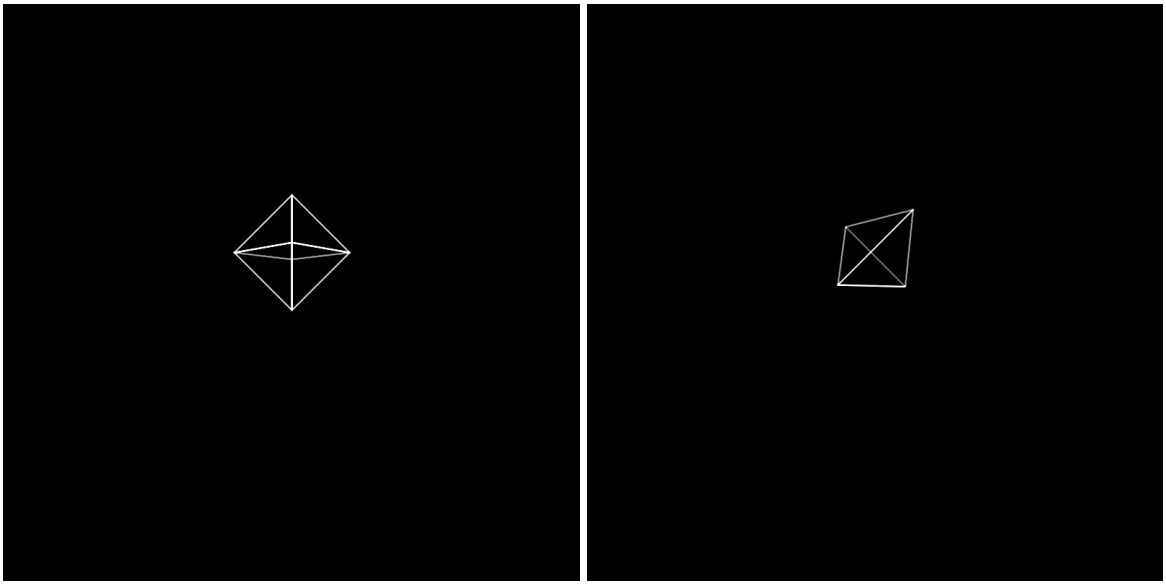
Main Rendering Loop:

Our implementation loops over 120 frames and in each frame, a time parameter *t* is calculated. A dynamic axis of rotation is derived using spherical linear interpolation (*vec3_slerp*) between two fixed axes. Local transformations are defined using *mat4_rotate_xyz* and *mat4_translate*, which together form the world transformation matrix. Finally, *render_wireframe()* is called to draw the transformed wireframe on the canvas.

Demo:



We rendered two more objects by applying the same transformations as of the soccer ball,



Task 4: Lightning and Polish

Introduction

As required, how the brightness value of each edge between vertices get affected according to implemented light sources has been primarily focused in this section. Specially, how the lightning sources are defined, implemented, support for multiple sources, projection onto the 2D screen has been taken into consideration.

Implementation details

Brightness value

The *lightning.c* file contains the functionalities used for assigning a brightness value for each edge in the wire frame.The main method that is implemented in here is named *void lightning()*

***lightning()*:**

This method takes several parameters as inputs and outputs a sorted list of edges[] that contains their respective brightness values. Edges in the context are a special structure element that is defined in this header file that contains

- The pair of vertices its drawn between (vec3 elements)
- Its relative distance measure to light sources
- The corresponding brightness value to be projected

As parameters this function primarily takes 3D virtual coordinates of the vertices, 2D list that describes on what vertices the lines are drawn between, an array that contains the coordinates of the light sources and some minor single values needed for operation. Its important to use the 3D coordinates rather than the projected ones because if the latter were used, the brightness values will be distorted improperly.

This function considers the midpoint of each edge or in other words the midpoint between vertices that a line is drawn between and measures the distance to each source and stores that values in an array of *edges_t.* This was done by iteratively measuring distance between each light source and the edges.

Then edges that are relatively further away from light sources are given a lower brightness value and those closer are given a higher value. When implementing this, the furthest and closest edges has been used as bounds for brightness distribution among edges, and relative to them and their relative distance to lightning sources was considered for each edge and then given an intensity

value. So an initial sorting has been performed to the wireframe edge to find furthest and closest elements accordingly.

Also, the grouping of fours edges has been done according to the relative distance to sources and given a the same intensity value to make a more contrasting appearance of brightness distribution during the rendering.

Apart from this function some minor functionality was also defined in this file for implementation ease and to avoid repetition in the code.

***compare_edges_by_distance()*:**

This function takes two pointer values to edge_t elements and compares their distance value and outputs whether the second is relatively close or not to the light sources. This was used to avoid repetition of code in sorting of edge list according to their light source distances.

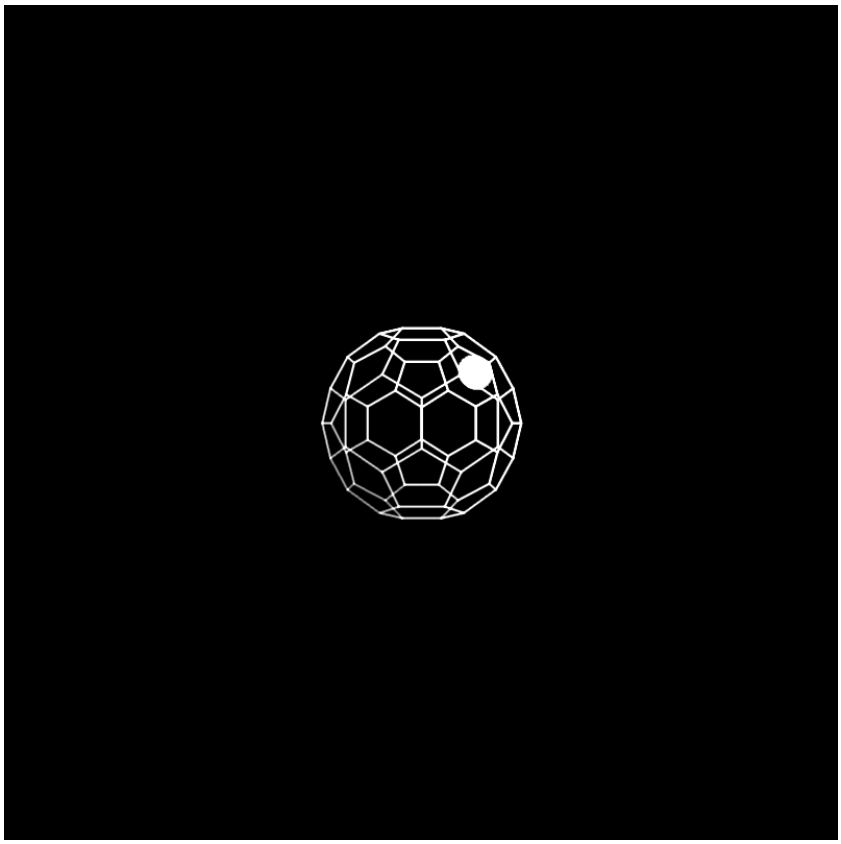
***map_value()*:**

map _value function was used to give a brightness value to each edge relative to the furthest and closest edge to light sources and their own relative distance as mentioned above.

Projecting light sources to the canvas

When light sources are projected to the canvas, simple orthogonal projection has been done rather than using projection methods used for lines and vertices considering the static nature of them in the canvas and how well it is suited given their behaviour.

Demo



Group_56

Group Members: Weerasinghe W.P.T.H. – E/22/421 – Tharindu
Kariyawasam K.P.W.D.R. – E/22/182 – Dinith

The whole entirety of the project was evenly contributed by both team members. A public github repository was maintained for the ease of sharing the codes and other significant files.

Contributions

Tharindu:

- **Task 1:** Entirety of the task was completed by Tharindu, including producing the demo clock as a JPEG image.
- **Task 4:** Entirety of the task was completed by Tharindu, including producing the demo .gif file with the light source
- Depth sorting and brightness assigning, rendering accordingly with minimal warping, object translation, improvements and bug fixing in Task 2
- Co-authored the Task 1 and Task 4 sections of the project report
- Provided Task 1 and Task 4 sections of the video presentation

Dinith:

- **Task 2:** Entirety of the task was completed by Dinith, including producing the demo cube as a .gif file
- **Task 3:** Entirety of the task was completed by Dinith, including producing the animated soccer ball as a .gif file
- Co-authored the Task 2 and Task 3 sections of the project report
- Provided Task 2 and Task 3 sections of the video presentation
- Looked into how to maintain and work with github repositories to ease the workflow of the project

AI Tool Usage

AI tools have been primarily used for finding proper coordinates of the objects to be rendered. However, the edges or vertex pairs to be matched had to be found manually through testing. For example, separate code had to be used for finding edges in the truncated icosahedron, which is attached here.

Apart from that, AI tools were used for discovering related mathematics and to have suggestions during bug fixing and also to generate some parts of the code.