3D Graphics Implementation using LPC1769

Divya Khandelwal, MS.
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 94303
E-mail: divya.khandelwal@sjsu.edu

Abstract

In this project, a graphic display prototype is developed using LPC 1769 interfaced with a 1.8” ST7735R TFT LCD. The aim is to display a 3D shading model and diffuse reflection computation on LPC1769 microprocessor platform. The key in this project is create a basic 3D Graphics Processing Engine.

1. Introduction

The LPC1769 is a Cortex-M3 microcontroller for embedded applications featuring a high level of integration and low power consumption at frequencies of 120MHz [1]. Along with numerous other features, this module has 70 General Purpose I/O (GPIO) pins with configurable pull up/down resistors and 3 SSP/SPI pins. For this project, these two functionalities of LPC1769 are the key.

For this project, a prototype circuit is created using LPC1769 module and 1.8” ST7735R TFT LCD. The ST7735R is a single-chip controller/driver for 262K-color, graphic type TFT-LCD [2]. It accepts Serial Peripheral Interface (SPI) when connected to an external microcontroller. The display data is stored in the on-chip display data RAM of 132 x 162 x 18 bits.

MCUXpresso from NXP has been used as the IDE to work with this module using C programming language.

2. Methodology

In this section, system layout and its hardware and software configurations are discussed. In addition, the objectives for this project have been listed with the challenges faced during implementation.

2.1. Objectives and Technical Challenges

The objectives of this project are as follows:

1. To be able to create a prototype circuit with LPC1769 module, power-supply and LCD.
2. To gain hands on experience with MCUXpresso environment, its debugging tools and C programming language.
3. To understand and implement SSP and SPI interface.

The technical challenges faced are as follows:

1. Soldering the components on the wire-wrapping board with proper pin connections.
2. Understanding the 3D vector graphics mathematics and implementing it in the C code.
3. Displaying the graphic pattern over the physical display using coordinate system mapping.

2.2. Problem Formulation and Design

In this section, we discuss the system design for the project. This project focuses on generating a solid cube with its shadow. A point light source is assumed in the world coordinate system and graphics have been created based on its position. Here, Fig.1 shows the system block diagram.

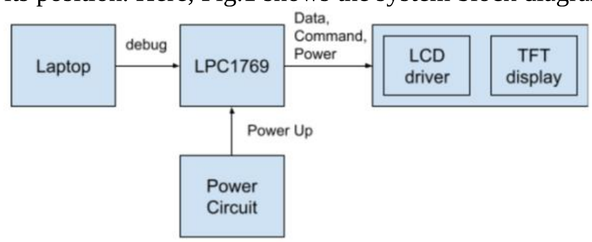

Fig.1 System Block Diagram

3. Implementation

In this section the hardware and software design of the project is described involving LPC1769 interfaced with TFT display.

3.1. Hardware Design

This section describes the list of components, system pin connectivity information between subsystems, schematic of the system and photos of the implemented project board.

3.1.1 List of Components

Table I. lists the hardware components required to build the prototype.

S.No	Item Name
1.	Wire Wrapping Board
2.	LPC1769 CPU module
	Power/Debug Cable
3.	160 x 128 TFT display (ST7735)
	Pin Headers
4.	9V wall mount Power Adaptor
	J1 right angle connector
	Capacitors (100uF/50V, 10uF/50V)
	LM7805 Voltage regulator
	LED (Red)
	SPDT Switch
	Resistor (330 ohms)
5.	Stand offs

Table I. List of Hardware Components

3.1.2 Serial Peripheral Interface

Serial Peripheral Interface is a full duplex serial interface. It can handle multiple masters and slaves being connected to a given bus. [3] During a data transfer the master always sends 8 to 16 bits of data to the slave, and the slave always sends a byte of data to the master. Fig. 2 shows the communication of master with its slaves over SPI [4]. Master uses slave select signal (SS) to select which slave it wants to communicate to.

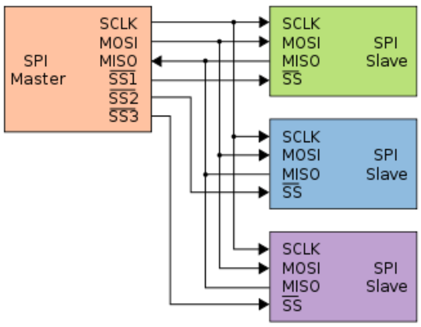

Fig. 2 Serial Peripheral Interface

3.1.3 System Schematic

The system works in Master Slave configuration. The LPC1769 acts as master which communicates with the slave i.e LCD display via a Serial Peripheral Interface (SPI). Fig.3 shows the pin connectivity of the two modules through a schematic diagram.

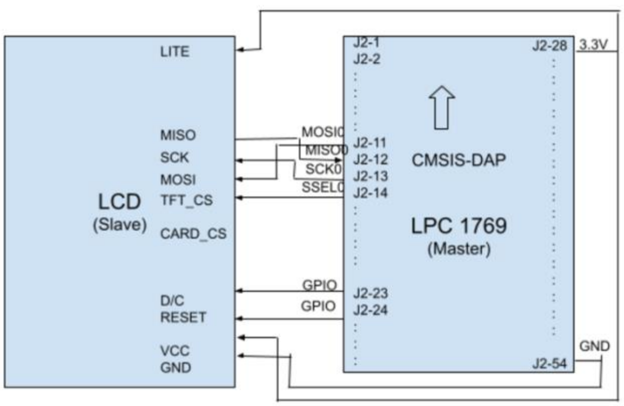

Fig.3 System Schematic

LCD Pins	LPC1769 ports	LPC1769 pins
LITE	VCC (3.3V)	J2-28
MISO	P0.17 (MISO0)	J2-12
SCK	P0.15 (SCK0)	J2-13
MOSI	P0.18 (MOSI0)	J2-11
TFT_CS	P0.16 (SSEL0)	J2-14
CARD_CS	-	-
D/C	P0.21(GPIO)	J2-23
RESET	P0.22 (GPIO)	J2-24
VCC	VCC (3.3V)	J2-28
GND	GND	J2-1/J2-54

Table II. Pin Connectivity Table

3.2. Software Design

This section focuses on the development environment (IDE) used, algorithm design and Pseudo code.

3.2.1 Integrated Development Environment (IDE)

In this project, MCUExpresso by NXP has been used as the IDE. MCUXpresso is a GNU and Eclipse-based IDE that provides an easy-to-use development environment for general purpose, crossover and Bluetooth Arm Cortex-M-based MCUs from NXP, here LPC1769.
For this project, C is used as the programming language and the code is operated in debug mode on the CPU module.

3.2.2 Algorithm Design

This project follows a basic system algorithm based on SPI interface between LPC1769 and the LCD display. The MCUXpresso also plays an important role since it allows the user to render an input to modify the graphic display, through the debug mode. Fig.4 shows the flow chart of the working system.

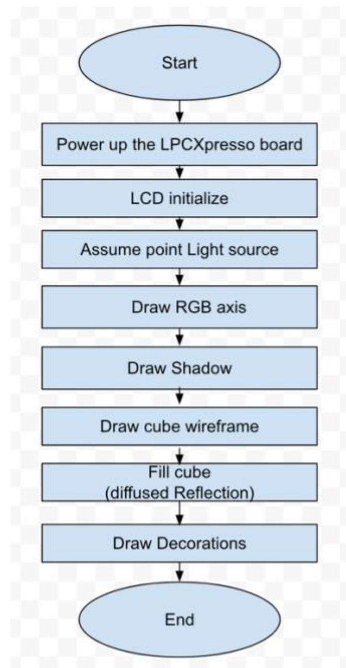

Fig.4 System Flow chart

In this project, it is required to read and write to various registers of the LPC1769 for interfacing with LCD display and successful data/command exchange. Table III shows the list of those 32-bit registers.

Used to make LPC1769 - GPIO pins	
LPC_GPOI -> FIODIR	Direction of the I/O (Input pin =0 / Output pin = 1)
LPC_GPIO -> FIOPIN	It reads value at the input pin
Used to select the slave – sends output to TFT_CS	
LPC_GPIO ->FIOSET	It sets the output pin
LPC_GPIO -> FIOCLR	It clears the output

Used to make P0.15-P0.18 of LPC1769 as SSP0	
LPC_PINCON->PINSEL0	Pin[31:30]=10; select SCK0
LPC_PINCON->PINSEL1	Pin[1:0]= 10; select SSEL0 Pin[3:2]= 10; select MISO0 Pin[5:4]= 10; select MOSI0
Used to control basic operations of SSP controller (here used to fetch SPI frame format)	
LPC_SSP0->CR0	Pin[3:0]=0111; select 8-bit transfer Pin[5:4]=00; SPI frame format Pin[6]=0/1; Clock out polarity Pin[7]=0/1; Clock out phase Pin[15:8]= Serial clock Rate Pin[31:15]= 0x00 ; Reserved

Table III. Registers modified for LPC1769 and LCD interface

3.2.3 Implementation – Graphic Design

The following are the milestones and key concepts of the 3D graphics project:

1. Point Light source:
   Assume point light source at some point in world coordinate system and the calculation will be done based on that.

2. RGB axis:
   The world coordinate system is in a 3D coordinate system of x-y-z axis. We use the concept of perspective projection to display it on a 2D screen.

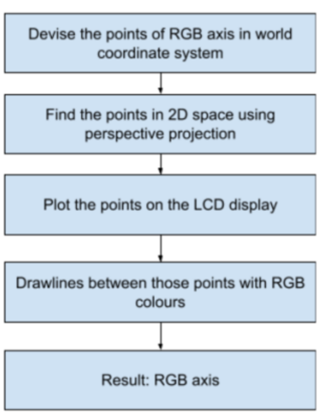

Fig.5 Process to generate RGB axis

The vector graphics formula used to do covert world to viewer coordinate system:
LHS = Viewer coordinate system
RHS = Transformation matrix & Word coordinate system

$$(Xw,\ Yw,\ Zw)\ \varepsilon\ R3\ \rightarrow\ (Xv,\ Yv,\ Zv)\ \varepsilon\ R3$$

$$Xv\ =\ -(sin\theta * Xw)\ +\ (cos\theta * Yw)$$

$$Yv\ =\ -(cos\phi * cos\theta)\ Xw\ +\ (-\ cos\phi * sin\theta)\ Yw\ +\ (sin\phi)\ Zw$$

$$Zv\ =\ -(sin\phi * cos\theta)\ Xw\ +(-\ sin\phi * cos\theta) Yw\ +\ (-\ cos\phi) Zw\ +\ \varrho$$

Perspective Projection:

$$(Xv,\ Yv,\ Zv)\ \varepsilon\ R3\ \rightarrow\ (Xp,\ Yp)\ \varepsilon\ R2$$

$$Xp\ =\ D/Zv\ *\ Xv$$

$$Yp\ =\ D/Zv\ *\ Yv$$

3. Solid Cube Generation:
   Expand the calculation done for RGB axis and find the vertices of the wireframe cube model. Then join those vertices using straight lines.

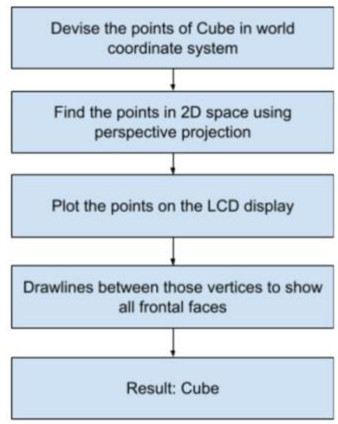

Fig.5 Process to generate solid cube

4. Shadow Generation:
   Shadow of the floating cube is generated with respect to the position of the point light source.

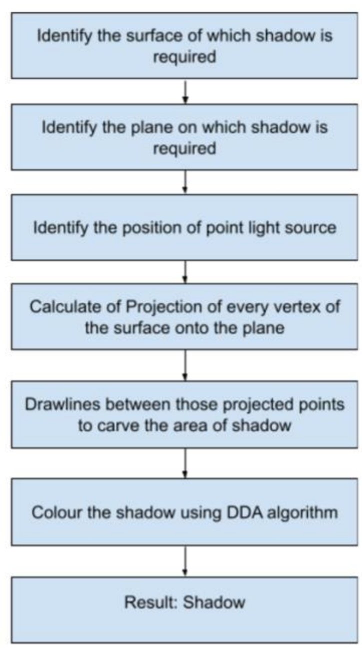

Fig.5 Process to generate shadow

The vertices of the shadow lie on the plane perpendicular to the axis of floating. In this project, the shadow appears at x-y plane. The vertices are calculated using the equation of a plane and the equation of a ray.
Equation of a plane:

$$\bar{n} \,.\, (\bar{v} - \bar{a}) = 0$$

Where n is the normal vector to the plane(known);
a is the arbitrary point on the plane (known);
v is another point on the plane (unknown)

Equation of the ray:

$$\overline{R} = \overline{Ps} + \lambda\,(\overline{Pi} - \overline{Ps})$$

Intersection of these two equations gives the shadow point. In the above equations, we have,

$$\bar{v} = \overline{R}$$

5. DDA algorithm:
This algorithm is used to calculate all points lying on a line segment whose starting and ending points are known. We also use a scanning line to evaluate all the interior points.

6. Diffused Reflection

Diffused Reflection occurs when light is reflected by an object in all direction. It is due to diffused reflection that the color of an object appears the way it is.
The equation of diffused reflection is as follows:

$$I(x,y,z) = [\,Kr * (\bar{n}.\bar{r}) / (\|\bar{n}\|.\,\|r\|)] / \|r\|2$$

where,
Kr = Reflectivity
n = normal vector
r = ray vector, which is $(\overline{Pi} - \overline{Ps})$

4. Testing and Verification

This section focuses on testing and verification of the prototype hardware and software so as to confirm that obtained results matches the desired ones.

4.1 Testing

The following are the test cases of hardware:

1. Make sure the Power plug is connected to the J1 power connector port.
2. Check for the power circuit switch to be in ON state and power circuit LED is glowing.
3. Check if all the soldered connection are intact and correct.
4. Make sure the program is run in debug mode on LPC1769 from MCUXpresso IDE.
5. Check if the desired graphics are rendered on the display.

The following are the test cases of software:

1. Make sure that calculated diffused reflection is multiplied with scaling factor > 10000 so as to see visible gradient on LCD display.
2. Add an offset of approx. 20 to the diffused reflection to make gradient distinguishable on the LCD display.

4.2 Verification

Below are the results of the implementation of the project:

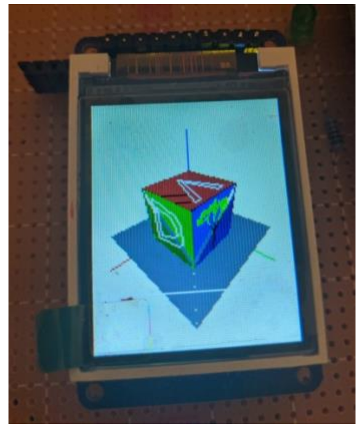

Fig.6 Solid cube with diffused reflection and decorations

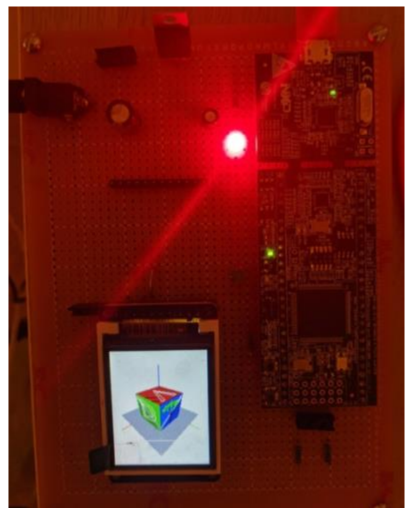

Fig.7 Hardware Assembly showing LPC1769, LCD module, GPIO circuit and Power circuit

5. Conclusion

The 3D graphic pattern of solid with shadow and diffused reflection was successfully rendered on the LCD display interfaced with LPC1769. In Section 8, the source code is attached (Appendix).

6. Acknowledgement

This project was successfully implemented under the mentorship of Dr. Harry Li, Department of Computer Engineering, San Jose State University, CA.

7. References

[1] H. Li, “2018F-107-lecGPP-2018-9-10”, *Lecture Notes of CMPE 242*, Computer Engineering Dept., College of Engineering, San Jose State University, September 10, 2018 pp. 1.
[2] Sitronix Technology Corp., “ST7735R 262K Color Single-Chip TFT Controller/Driver”, V0.2, May 8, 2009.

[3] NXP Inc., “LPC 1769/68/67/66/65/64/63 data sheet”, Revision 9.10, September 8, 2020
[4] en.wikipedia.org/wiki/Serial_Peripheral_Interface

8. Appendix

Below is the source code of the C program for 3D vector graphics.

```
/*
 * main.c
 *
 *  Created on: May 12, 2021
 *      Author: divya
 */

#include "Divya_Lab3D_CMPE240.h"
#include "globals.h"

int main (void){
 uint32_t pnum = PORT_NUM;
 pnum = 0 ;
 if ( pnum == 0 )       SSP0Init();
 else   puts("Port number is not correct");
 lcd_init();
 fillrect(0, 0, ST7735_TFTWIDTH, ST7735_TFTHEIGHT,
LIGHTYELLOW);
 int ll = 100;
 int ll_ = 110;
 int _ll = 10;
 int world_pt = 180;
 point3D Pw[4] = {{0, 0, 0},{world_pt, 0, 0},{0,
world_pt, 0},{0, 0, world_pt}};
 point3D p1 = {0,ll,ll_};
 point3D p2 = {0,0,ll_};
 point3D p3 = {ll,0,ll_};
 point3D p4 = {ll,ll,ll_};
 point3D p5 = {ll,ll,_ll};
 point3D p6 = {0,ll,_ll};
 point3D p7 = {0,0,_ll};
 point3D p8 = {ll,0,_ll};
 point3D Vertex_cube[8] = {p1, p2, p3, p4, p5, p6,
p7, p8};
 point2D physical_vertices_cube[8];
 point3D normal_xy_plane ={0,0,1};
 point3D normal_xz_plane ={0,1,0};
 point3D normal_yz_plane ={1,0,0};
 point3D arbitrary_point = {0,0,0};

drawRGBaxis(Pw);
makeShadow( Vertex_cube , 4, normal_xy_plane,
arbitrary_point, GRAY );
drawCube( Vertex_cube, 8, BLACK);
//COLOR THE TOP FACE OF CUBE
point3D top_face_cube[4] = {p1, p2, p3, p4};
fillPolygonGradient(top_face_cube, 4,
normal_xy_plane, REFLECTIVITY_R);

//COLOR THE FRONTAL FACE (RIGHT) OF CUBE
point3D right_frontal_face_cube[4] = {p5, p6, p1,
p4};
fillPolygonGradient(right_frontal_face_cube, 4,
normal_xz_plane, REFLECTIVITY_B);

//COLOR THE FRONTAL FACE (LEFT) OF CUBE
point3D left_frontal_face_cube[4] = {p3, p4, p5,
p8};
fillPolygonGradient(left_frontal_face_cube, 4,
normal_yz_plane, REFLECTIVITY_R);
// ---------------------
//      create tree
//----------------------
 point3D root = {0.1*ll_, 0.5*ll, ll};
 point3D head = {0.6*ll_, 0.5*ll,ll};
 draw_complete_tree_3d(root, head);
// ---------------------
//      create letter D
// ---------------------
 point3D outer_D[6] = {{ll, 0.2*ll, 0.2*ll_} ,{ll,
0.2*ll, 0.9*ll_} ,{ll, 0.7*ll, 0.8*ll_} ,{ll,
0.8*ll, 0.7*ll_} ,{ll, 0.8*ll, 0.4*ll_},{ll,
0.7*ll, 0.3*ll_}};

 point3D inner_D[6] = {{ll, 0.3*ll, 0.3*ll_} ,{ll,
0.3*ll, 0.8*ll_} ,{ll, 0.6*ll, 0.7*ll_},{ll,
0.7*ll, 0.6*ll_},{ll, 0.7*ll, 0.5*ll_},{ll,
0.6*ll, 0.4*ll_}};
 for(int i = 0; i < 5; i++ ){
        drawLine3D(outer_D+i, outer_D+i+1,
WHITE);
        drawLine3D(inner_D+i, inner_D+i+1,
WHITE);
 }
 drawLine3D(outer_D+5, outer_D, WHITE);
 drawLine3D(inner_D+5, inner_D, WHITE);
 // ---------------------
 //      create letter V
 // ---------------------
 point3D V[6] = {{0.5*ll, 0.2*ll, ll_},{0.2*ll,
0.9*ll, ll_},{0.3*ll, 0.9*ll, ll_},{0.5*ll,
0.3*ll, ll_} ,{0.7*ll, 0.9*ll, ll_},{0.8*ll,
0.9*ll, ll_}};

 for(int i = 0; i < 5; i++ ){
        drawLine3D(V+i, V+i+1, WHITE);
 }
 drawLine3D(V+5, V, WHITE);

 return 0;
}


/*
 * Divya_Lab3D_CMPE240.c
 *
 *  Created on: May 12, 2021
 *      Author: divya
 */


#include "Divya_Lab3D_CMPE240.h"
#include <stdio.h>

//------------------ Diffused reflection of the
vertices of desired surface- Id (r,g,b)  ---------
-------//
RGBdiffusedReflection
diffusedReflectionVertexForRGBColors(const point3D
normal_vector,const RGBreflectivity reflectivity,
               const point3D surface_point){
       RGBdiffusedReflection Id;
       Id.r =
diffusedReflectionVertexForOneColor(
normal_vector, reflectivity.r, surface_point );
       Id.g =diffusedReflectionVertexForOneColor(
normal_vector, reflectivity.g, surface_point );
       Id.b =diffusedReflectionVertexForOneColor(
normal_vector, reflectivity.b, surface_point );
       return Id;
}

float diffusedReflectionVertexForOneColor(const
point3D normal_vector, const float reflectivity,
               const point3D surface_point){
       float diffused_reflection, angle;
       point3D ray =
subtract(POINT_LIGHT_SOURCE,surface_point);
```

```
        angle = dotProduct(normal_vector,ray) /
(modulus(ray) * modulus(normal_vector));
        diffused_reflection = angle /
pow(modulus(ray),2);
        diffused_reflection = diffused_reflection
* SCALING_FACTOR + OFFSET;
        return reflectivity * diffused_reflection;
}

//------------------ Fill a ploygon with Gradient
colours  ----------------//
void fillPolygonGradient(const point3D*
vertices_surface, const int size, const point3D
normal_vector,
                const RGBreflectivity
reflectivity){
        //-----------diffused Reflection For
Vertices--------------------//
                RGBdiffusedReflection Id[4];
                uint32_t color[4];
                for(int i = 0; i < 4; i++){
                        Id[i] =
diffusedReflectionVertexForRGBColors(normal_vector
, reflectivity, vertices_surface[i]);
                        color[i] =
findcolor(Id[i]);
                        drawPixel3D(
vertices_surface + i, color[i]);
                }
        //-----------colour the boundary points of
a polygon--------------------------//
                point2D physical_vertices[size];
                line shape_edges[size];
                RGBdiffusedReflection point_Id;
                uint32_t temp_color;

        convert3Dto2Dpoints(physical_vertices,
vertices_surface, size);
                //-----------colour the entire
polygon with gradient--------------------------//
                polygon face_polygon;
                face_polygon.size = size;
                //allocating size to pointer of a
line
                face_polygon.edge = (line*)
malloc(sizeof(int) + sizeof(point2D*));
                face_polygon.edge->point =
(point2D*) malloc(2 * sizeof(int));
                face_polygon.vertex =
physical_vertices;
                gradientUsingDDA(&face_polygon, Id,
size);
}

uint32_t findcolor(RGBdiffusedReflection Id){
        uint32_t color = 0;
        float red,  green, blue;
        red    = (255 * Id.r);
        green = (255 * Id.g);
        blue = (255 * Id.b);
        color = (((uint32_t)red) << 16) |
(((uint32_t)green) << 8) | ((uint32_t)blue);
        return color;
}
void
fillaBoundaryLineWithGradient(RGBdiffusedReflectio
n* point_Id, const line* l, const point2D
start_pt, const point2D end_pt,
        const RGBdiffusedReflection
start_pt_diffused_reflection, const
RGBdiffusedReflection end_pt_diffused_reflection){
        uint32_t temp_color;

        for(int i = 0; i < l->size ; i++ ){
                point_Id[i] =
diffusedReflectionOfPointOnALine(start_pt, end_pt,
                        l->point[i],
start_pt_diffused_reflection,
end_pt_diffused_reflection);
                temp_color =
findcolor(point_Id[i]);
                drawPixel( l->point[i].x , l-
>point[i].y , temp_color);
        }
}
void fillaInteriorLineWithGradient(const line* l,
const point2D start_pt, const point2D end_pt,
        const RGBdiffusedReflection
start_pt_diffused_reflection, const
RGBdiffusedReflection end_pt_diffused_reflection){
        uint32_t temp_color;
        RGBdiffusedReflection point_Id;
        for(int i = 0; i < l->size ; i++ ){
                point_Id =
diffusedReflectionOfPointOnALine(start_pt, end_pt,
                        l->point[i],
start_pt_diffused_reflection,
end_pt_diffused_reflection);
                temp_color = findcolor(point_Id);
                drawPixel( l->point[i].x , l-
>point[i].y , temp_color);
        }
}

//-------------------- Diffused reflection of any
point on a boundary line  ------------------------
-----------//

RGBdiffusedReflection
diffusedReflectionOfPointOnALine(const point2D
start_pt, const point2D end_pt,
                const point2D arbitrary_point,
RGBdiffusedReflection
start_pt_diffused_reflection,
                RGBdiffusedReflection
end_pt_diffused_reflection){
        RGBdiffusedReflection Id;
        Id.r =
diffusedReflectionIntermediatePointOneColor(start_
pt, end_pt, arbitrary_point,
start_pt_diffused_reflection.r ,
                end_pt_diffused_reflection.r);
        Id.g =
diffusedReflectionIntermediatePointOneColor(start_
pt, end_pt, arbitrary_point,
start_pt_diffused_reflection.g,
                end_pt_diffused_reflection.g);
        Id.b =
diffusedReflectionIntermediatePointOneColor(start_
pt, end_pt, arbitrary_point,
start_pt_diffused_reflection.b ,
                end_pt_diffused_reflection.b);
        return Id;
}

float
diffusedReflectionIntermediatePointOneColor(const
point2D start_pt, const point2D end_pt, const
point2D arbitrary_point,
        float start_pt_diffused_reflection,  float
end_pt_diffused_reflection){
        float Idx, Idy;
        Idx =
diffusedReflectionOfPointCoordinate(start_pt.x,
end_pt.x, arbitrary_point.x,

        start_pt_diffused_reflection,
end_pt_diffused_reflection);
```

```
	Idy =
diffusedReflectionOfPointCoordinate(start_pt.y,
end_pt.y, arbitrary_point.y,

	start_pt_diffused_reflection,
end_pt_diffused_reflection);
	 return 0.5 * (Idx + Idy);
}

float diffusedReflectionOfPointCoordinate(const
int start_pt_coordinate, const int
end_pt_coordinate,
		const int
arbitrary_point_coordinate, float
start_pt_diffused_reflection, float
end_pt_diffused_reflection){
	float exp1, exp2, exp3;
	exp1 = start_pt_diffused_reflection -
end_pt_diffused_reflection;
	exp2 = (arbitrary_point_coordinate -
end_pt_coordinate) / (start_pt_coordinate -
end_pt_coordinate);
	exp3 = end_pt_diffused_reflection;
	return exp1*exp2 + exp3;
}

//-------------------- DDA algorithm to calculate
all points on a line and color them --------------
---------------------//

void* pointsOnALine(line* l, const point2D
start_pt, const point2D end_pt){
	double delta_x, delta_y, number_of_steps;
	float increment_x, increment_y;
	float curr_pt_x = start_pt.x;
	float curr_pt_y = start_pt.y;
	delta_x =  (end_pt.x - start_pt.x);
	delta_y =  (end_pt.y - start_pt.y);
	number_of_steps = maximum(abs(delta_x),
abs(delta_y));
	l->size = (int)number_of_steps;
	//dynamically allocating memory to store
points in a line
	l->point = malloc(l->size *
sizeof(point2D));
	increment_x = delta_x /
(float)number_of_steps;
	increment_y = delta_y /
(float)number_of_steps;
	//Each iteration calculates next point on
line
	for(int i = 0; i < l->size  ; i++ ){
		curr_pt_x += increment_x;
		curr_pt_y += increment_y;
		l->point[i].x = (int)
round_value(curr_pt_x);
		l->point[i].y = (int)
round_value(curr_pt_y);
	}
	return l->point;
}

//-------------colour a polygon using  Scanning
line and DDA algorithm-------------------//

void fillPolygon(polygon *my_polygon, const
uint32_t color){
	int poly_size = my_polygon -> size;
	point2D start_pt, end_pt;
	line scanning_line;
	int first_edge_size;

	//-----calculate points of first 2 edges--
---//
	void* free_karne_waali_line[2];
	for(int i = 0 ; i < poly_size/2 ; i++){
		free_karne_waali_line[i] =
pointsOnALine(my_polygon->edge+i, my_polygon ->
vertex[i],
					my_polygon ->
vertex[i+1]);
		fillaLine(my_polygon->edge+i,
color);
	}

	first_edge_size = my_polygon-
>edge[0].size;
	void* temp;
	for(int j = 0 ; j < first_edge_size ;
j++){
		start_pt = my_polygon ->
edge[0].point[first_edge_size-j-1];
		end_pt = my_polygon ->
edge[1].point[j];
		temp =
pointsOnALine(&scanning_line, start_pt, end_pt);
		fillaLine(&scanning_line, color);
		free(temp);
	}
	free(free_karne_waali_line[0]);
	free(free_karne_waali_line[1]);
	free(my_polygon -> edge);
	free(my_polygon -> edge+1);

	//points of 3rd and 4th edge
	void* free_karne_waali_line2[2];
	free_karne_waali_line2[0]=pointsOnALine(my
_polygon->edge + 2, my_polygon -> vertex[2],
			my_polygon -> vertex[3]);
	fillaLine(my_polygon ->edge + 2, color);
	free_karne_waali_line2[1]=pointsOnALine(my
_polygon->edge+poly_size-1, my_polygon ->
vertex[poly_size-1],
					my_polygon ->
vertex[0]);
	fillaLine(my_polygon ->edge + poly_size-1,
color);


	first_edge_size = my_polygon-
>edge[2].size;
	int second_edge_size = my_polygon-
>edge[3].size;
	for(int j = 0 ; j < first_edge_size ;
j++){
		start_pt = my_polygon ->
edge[2].point[j];
		end_pt = my_polygon ->
edge[3].point[my_polygon->edge[2].size - j -1];
		temp =
pointsOnALine(&scanning_line, start_pt, end_pt);
		fillaLine(&scanning_line, color);
		free(temp);
	}

	free(free_karne_waali_line2[0]);
	free(free_karne_waali_line2[1]);
	free(my_polygon -> edge + 2);
	free(my_polygon -> edge + 3);

}

void fillaLine(const line* l, uint32_t color){
	for(int i = 0; i < l->size ; i++ ){
			drawPixel( l->point[i].x ,
l->point[i].y , color);
	}
}

float maximum(const float a, const float b){
```

```
	if(b > a) return b;
	else return a;
}

float round_value(float v){
  return floor(v + 0.5);
}

//-------------------- Make coloured shadow of
surface of cube ---------------------------------
-//

void makeShadow( const point3D *Vertices_surface,
const int size, const point3D normal_vector,
	const point3D arbitrary_point, uint32_t
color ){
	float lambda[size];
	point3D intersection_point[size];
	point2D
physical_instersection_points[size];
//	printf("Following are the shadow points:
\n");
	for(int i = 0; i < size; i++){
		lambda[i] =
findLambda(Vertices_surface[i], normal_vector,
arbitrary_point);
		intersection_point[i] =
findinstersectionPointin3D(Vertices_surface[i],lam
bda[i]);
	}
	for(int i = 0 ; i < size-1 ; i++){
		drawLine3D(intersection_point +
i,intersection_point + i + 1,BLACK);
	}
	drawLine3D(intersection_point + size - 1,
intersection_point,BLACK);
	printPoint3D(intersection_point[0]);

	printPoint3D(intersection_point[1]);

	printPoint3D(intersection_point[2]);

	printPoint3D(intersection_point[3]);
	//fetch the physical coordinates of shadow
to use to fill it
	convert3Dto2Dpoints(physical_instersection
_points, intersection_point, size);
	polygon shadow_polygon;
	shadow_polygon.size = size;

	//allocating size to pointer of a line
	shadow_polygon.edge = (line*)
malloc(sizeof(int) + sizeof(point2D*));
	shadow_polygon.edge->point = (point2D*)
malloc(2 * sizeof(int));
	shadow_polygon.vertex =
physical_instersection_points;

	fillPolygon(&shadow_polygon, color);

}

void gradientUsingDDA(polygon *my_polygon,
RGBdiffusedReflection* Id, int size){
	int poly_size = my_polygon -> size;
	point2D start_pt, end_pt;
	line scanning_line;
	int first_edge_size;
	RGBdiffusedReflection* point_Id[2];

	//-----calculate points of first 2 edges--
---//
	void* free_karne_waali_line[2];
	for(int i = 0 ; i < poly_size/2 ; i++){
		free_karne_waali_line[i] =
pointsOnALine(my_polygon->edge+i, my_polygon ->
vertex[i],
					my_polygon ->
vertex[i+1]);
		point_Id[i] =
(RGBdiffusedReflection*) malloc(my_polygon-
>edge[i].size * 3 * sizeof(float));

	fillaBoundaryLineWithGradient(point_Id[i],
my_polygon->edge+i,  my_polygon -> vertex[i],
my_polygon -> vertex[i+1],
					Id[i], Id[i+1]);
	}

	first_edge_size = my_polygon-
>edge[0].size;
	void* temp;
	for(int j = 0 ; j < first_edge_size ;
j++){
		start_pt = my_polygon ->
edge[0].point[first_edge_size-j-1];
		end_pt = my_polygon ->
edge[1].point[j];
		temp =
pointsOnALine(&scanning_line, start_pt, end_pt);

	fillaInteriorLineWithGradient(&scanning_li
ne,  start_pt, end_pt,

	point_Id[0][first_edge_size-j-1],
point_Id[1][j]);
		free(temp);
	}
	free(point_Id[0]);
	free(point_Id[1]);
	free(free_karne_waali_line[0]);
	free(free_karne_waali_line[1]);
	free(my_polygon -> edge);
	free(my_polygon -> edge+1);

	//points of 3rd and 4th edge
	void* free_karne_waali_line2[2];
	free_karne_waali_line2[0]=pointsOnALine(my
_polygon->edge + 2, my_polygon -> vertex[2],
			my_polygon -> vertex[3]);
	point_Id[0] = (RGBdiffusedReflection*)
malloc(my_polygon->edge[2].size * 3 *
sizeof(float));
	fillaBoundaryLineWithGradient(point_Id[0],
my_polygon->edge+2,  my_polygon -> vertex[2],
my_polygon -> vertex[3],
					Id[2], Id[3]);
	free_karne_waali_line2[1]=pointsOnALine(my
_polygon->edge+poly_size-1, my_polygon ->
vertex[poly_size-1],
					my_polygon ->
vertex[0]);
	point_Id[1] = (RGBdiffusedReflection*)
malloc(my_polygon->edge[poly_size-1].size * 3 *
sizeof(float));
	fillaBoundaryLineWithGradient(point_Id[1],
my_polygon->edge+poly_size-1,  my_polygon ->
vertex[poly_size-1],
			my_polygon -> vertex[0],
Id[3], Id[0]);

	first_edge_size = my_polygon-
>edge[2].size;
	 int second_edge_size = my_polygon-
>edge[3].size;
	 for(int j = 0 ; j < first_edge_size ;
j++){
		start_pt = my_polygon ->
edge[2].point[j];
```

```
                end_pt = my_polygon ->
edge[3].point[my_polygon->edge[2].size - j -1];
                temp =
pointsOnALine(&scanning_line, start_pt, end_pt);

        fillaInteriorLineWithGradient(&scanning_li
ne,  start_pt, end_pt, point_Id[0][j],
                        point_Id[1][first_edge_size
- j -1]);
                free(temp);
         }
         free(point_Id[0]);
         free(point_Id[1]);
         free(free_karne_waali_line2[0]);
         free(free_karne_waali_line2[1]);
         free(my_polygon -> edge + 2);
         free(my_polygon -> edge + 3);

}

void colorAnyPolygon(const point3D normal_vector,
const point3D arbritary_point, const point3D
*Vertices_surface,
                const int size, const uint32_t
color){

        point2D physical_points[size];
        convert3Dto2Dpoints(physical_points,
Vertices_surface, size);
        polygon face_polygon;
        face_polygon.size = size;
        //allocating size to pointer of a line
        face_polygon.edge = (line*)
malloc(sizeof(int) + sizeof(point2D*));
        face_polygon.edge->point = (point2D*)
malloc(2 * sizeof(int));
        face_polygon.vertex = physical_points;
        fillPolygon(&face_polygon, color);
}


float findLambda( const point3D surface_vertex,
const point3D normal_vector, const point3D
arbritary_point ){
        float numerator, denominator;
        numerator = normal_vector.x *
(arbritary_point.x - POINT_LIGHT_SOURCE.x)
                                + normal_vector.y *
(arbritary_point.y - POINT_LIGHT_SOURCE.y)
                                + normal_vector.z *
(arbritary_point.z - POINT_LIGHT_SOURCE.z);
        denominator = normal_vector.x *
(surface_vertex.x - POINT_LIGHT_SOURCE.x)
                                + normal_vector.y *
(surface_vertex.y - POINT_LIGHT_SOURCE.y)
                                + normal_vector.z *
(surface_vertex.z - POINT_LIGHT_SOURCE.z);
        return numerator/denominator;
}

point3D findinstersectionPointin3D( const point3D
surface_vertex, float lambda ){
        point3D intersection_point;
        intersection_point.x =
POINT_LIGHT_SOURCE.x + lambda * (surface_vertex.x
- POINT_LIGHT_SOURCE.x);
        intersection_point.y =
POINT_LIGHT_SOURCE.y + lambda * (surface_vertex.y
- POINT_LIGHT_SOURCE.y);
        intersection_point.z =
POINT_LIGHT_SOURCE.z + lambda * (surface_vertex.z
- POINT_LIGHT_SOURCE.z);
        return intersection_point;
}
```

```
void drawRGBaxis(const point3D* axis_coordinates){
        drawLine3D( axis_coordinates,
axis_coordinates+1, RED );
        drawLine3D( axis_coordinates,
axis_coordinates+2, GREEN );
        drawLine3D( axis_coordinates,
axis_coordinates+3, BLUE );
}
void drawCube(const point3D* Vertex_cube, const
int size, uint32_t color){
        //DRAW TOP FACE
        drawLine3D(Vertex_cube,Vertex_cube+1,color
);
        drawLine3D(Vertex_cube+1,Vertex_cube+2,col
or);
        drawLine3D(Vertex_cube+2,Vertex_cube+3,col
or);
        drawLine3D(Vertex_cube+3,Vertex_cube+0,col
or);
        //DRAW BOTTOM FACE
        drawLine3D(Vertex_cube+4,Vertex_cube+5,col
or);
        drawLine3D(Vertex_cube+7,Vertex_cube+4,col
or);
        //DRAW VERTICAL EDGES
        drawLine3D(Vertex_cube+0,Vertex_cube+5,col
or);
        drawLine3D(Vertex_cube+2,Vertex_cube+7,col
or);
        drawLine3D(Vertex_cube+3,Vertex_cube+4,col
or);
}


void convert3Dto2Dpoints(point2D* physical_point,
const point3D *three_d_points, const int size){
        transformationMatrix t_matrix =
determineTranformationMatrix();
         for(int i=0; i<size; i++){
                physical_point[i] =
ThreeDTransformationPipelining(three_d_points[i],
t_matrix);
        }
}

point2D ThreeDTransformationPipelining(const
point3D Pw, const transformationMatrix t_matrix){
        point2D projected_point, physical_point;
        point3D Pv;
        Pv =
convertWorldCoordinatesToViewerCoordinates(Pw,
t_matrix);
        projected_point =
convertViewerCoordinatesToProjectedCoordinates(Pv)
;
        physical_point =
convertVirtualCoordinatesToPhysicalCoordinates(pro
jected_point);
        return physical_point;
}

transformationMatrix
determineTranformationMatrix(){
        transformationMatrix t_matrix;
        //Transformation matrix
        t_matrix.XY_HYPOTENOUS = sqrt
(pow(VIRTUAL_CAMERA.x , 2) + pow(VIRTUAL_CAMERA.y
, 2));
        t_matrix.RHO = sqrt ( pow(VIRTUAL_CAMERA.x
, 2) + pow(VIRTUAL_CAMERA.y , 2) +
pow(VIRTUAL_CAMERA.z , 2) );
        t_matrix.sin_THETA = VIRTUAL_CAMERA.y /
t_matrix.XY_HYPOTENOUS;
        t_matrix.cos_THETA = VIRTUAL_CAMERA.x /
t_matrix.XY_HYPOTENOUS;
```

```
	t_matrix.sin_PHI = t_matrix.XY_HYPOTENOUS
/ t_matrix.RHO;
	t_matrix.cos_PHI = VIRTUAL_CAMERA.z /
t_matrix.RHO;
	return t_matrix;
}

point3D
convertWorldCoordinatesToViewerCoordinates(const
point3D Pw, const transformationMatrix t_matrix){
	point3D Pv;
	Pv.x = (-1 * t_matrix.sin_THETA * Pw.x) +
(t_matrix.cos_THETA * Pw.y);
	Pv.y = (-1 * t_matrix.cos_PHI *
t_matrix.cos_THETA * Pw.x) + (-1 *
t_matrix.cos_PHI * t_matrix.sin_THETA * Pw.y) +
(t_matrix.sin_PHI * Pw.z);
	Pv.z = (-1 * t_matrix.sin_PHI *
t_matrix.cos_THETA * Pw.x) + (-1 *
t_matrix.sin_PHI * t_matrix.cos_THETA * Pw.y) + (-
1 * t_matrix.cos_PHI * Pw.z) + (t_matrix.RHO) ;
	return Pv;
}

point2D
convertViewerCoordinatesToProjectedCoordinates(con
st point3D Pv){
	point2D projected_point;
	projected_point.x = FOCAL_LENGTH_D * Pv.x
/ Pv.z;
	projected_point.y = FOCAL_LENGTH_D * Pv.y
/ Pv.z ;
	return projected_point;
}

point2D
convertVirtualCoordinatesToPhysicalCoordinates(con
st point2D projected_point){
	point2D physical_point;
	physical_point.x = projected_point.x +
(ST7735_TFTWIDTH/2);
	physical_point.y = -projected_point.y +
(ST7735_TFTHEIGHT/2);
	return physical_point;
}

float dotProduct(point3D p1, point3D p2){
	return ((p1.x * p2.x) + (p1.y * p2.y) +
(p1.z * p2.z));
}

float modulus(point3D p1){
	return sqrt(pow(p1.x,2) + pow(p1.y,2) +
pow(p1.z, 2));
}

point3D subtract(point3D p1, point3D p2){
	point3D result;
	result.x = p1.x - p2.x;
	result.y = p1.y - p2.y;
	result.z = p1.z - p2.z;
	return result;
}

void printPoint2D(point2D point){
	printf("{ %d, %d }\n", point.x , point.y);
}

void printPoint3D(point3D point){
	printf("{ %d, %d, %d}\n", point.x ,
point.y, point.z);
}

void printTranformationMatrix(transformationMatrix
t_matrix){
	printf("{ RHO, %f }\n", t_matrix.RHO);
	printf("{ sin_THETA:  %f }\n",
t_matrix.sin_THETA);
	printf("{ cos_THETA:  %f }\n",
t_matrix.cos_THETA);
	printf("{ sin_PHI:  %f }\n",
t_matrix.sin_PHI);
	printf("{ cos_PHI: %f }\n",
t_matrix.cos_PHI);
}


/*
 * DecorationShapes.c
 *
 *  Created on: May 12, 2021
 *      Author: divya
 */

#include "DecorationShapes.h"
#include "globals.h"
#define LAMBDA 0.6
#define CLK 0.52

void drawPixel3D(point3D* p0, uint32_t color){
	point2D physical_point[1];
	point3D three_d_points[1];
	three_d_points[0] = *p0;
	convert3Dto2Dpoints( physical_point,
three_d_points, 1 );

	drawPixel(physical_point[0].x,
physical_point[0].y, color);
}
void drawLine3D(point3D* p0, point3D* p1, uint32_t
color){
	point2D physical_point[2];
	point3D three_d_points[2];
	three_d_points[0] = *p0;
	three_d_points[1] = *p1;
	convert3Dto2Dpoints(physical_point,
three_d_points, 2);
	drawLine(physical_point[0].x,
			physical_point[0].y,
			physical_point[1].x,
			physical_point[1].y,
color);
}
void drawLine3D_jaadu(point3D* p0, point3D* p1,
uint32_t color){
	point2D physical_point[2];
	point3D three_d_points[2];

	// magic starts here
	three_d_points[0].z = p0->x;
	three_d_points[0].x = p0->y;
	three_d_points[0].y = p0->z; // constant
	three_d_points[1].z = p1->x;
	three_d_points[1].x = p1->y;
	three_d_points[1].y = p1->z; // constant
	// magic ends here

	convert3Dto2Dpoints(physical_point,
three_d_points, 2);
	drawLine(physical_point[0].x,
			physical_point[0].y,
			physical_point[1].x,
			physical_point[1].y,
color);
}
void tree_maker3D(float angle, struct tree3D
*base_tree, struct tree3D * right_tree, struct
tree3D * centre_tree,
		struct tree3D * left_tree){
```

```
        left_tree->p0 = base_tree->left_pt;
        left_tree->p1 =
branchExtension3D(&(base_tree->reduced_pt),
&(base_tree->left_pt), LAMBDA);
        drawLine3D_jaadu(&(left_tree->p0),
&(left_tree->p1), GREEN);
        drawHands3D(&(left_tree -> p0),
&(left_tree -> p1),LAMBDA, angle, GREEN,
&(left_tree->left_pt),&(left_tree-
>reduced_pt),&(left_tree->right_pt));

        centre_tree->p0 = base_tree->p1;
        centre_tree->p1 =
branchExtension3D(&(base_tree->reduced_pt),
&(base_tree->p1), LAMBDA);
        drawLine3D_jaadu(&(centre_tree-
>p0),&(centre_tree->p1), GREEN);
        drawHands3D(&(centre_tree -> p0),
&(centre_tree -> p1),LAMBDA, angle, GREEN,
&(centre_tree->left_pt),&(centre_tree-
>reduced_pt),&(centre_tree->right_pt));

        right_tree->p0 = base_tree->right_pt;
        right_tree->p1 =
branchExtension3D(&(base_tree->reduced_pt),
&(base_tree->right_pt), LAMBDA);
        drawLine3D_jaadu(&(right_tree-
>p0),&(right_tree->p1), GREEN);
        drawHands3D(&(right_tree -> p0),
&(right_tree -> p1),LAMBDA, angle, GREEN,
&(right_tree->left_pt),&(right_tree-
>reduced_pt),&(right_tree->right_pt));

}
void draw_complete_tree_3d(point3D p0, point3D
p1){
        int variance = 10*3.14/180;
        float rand_angle = CLK;// +
rand()%(variance*2) - variance;

        point3D p1_left, p1_right, reduced_point,
temp1,  temp2,  temp3,  temp4,  temp5,  temp6;
        drawLine3D_jaadu(&p0,&p1,BROWN);
        drawHands3D(&p0,&p1,LAMBDA, rand_angle,
BROWN, &p1_left, &reduced_point,&p1_right);

        struct tree3D base_tree;
    base_tree.p0= p0;
    base_tree.p1= p1;
    base_tree.right_pt = p1_right;
        base_tree.left_pt = p1_left;
        base_tree.reduced_pt= reduced_point;

        int max = 200;
        struct tree3D tree_array[max]; // 3^10
        tree_array[0]=base_tree;
        for(int i =0, j = 1; i<max && j <max;
i++){
        tree_maker3D(rand_angle, tree_array + i,
tree_array+j, tree_array+j+1, tree_array+j+2);
        j += 3;
        }
}
void drawHands3D(point3D *p0,point3D  *p1, float
lambda, float angle, uint32_t color,
               point3D  *p1_left, point3D
*reduced_point,point3D  *p1_right){
        // draw right branch
        calculateBranchEnds3D(p0, p1 ,angle,
lambda, reduced_point, p1_right);
        drawLine3D_jaadu(reduced_point,p1_right,
color);

        // draw left branch
        calculateBranchEnds3D(p0, p1 ,-angle,
lambda, reduced_point, p1_left);
        drawLine3D_jaadu(reduced_point,p1_left,
color);
}


void calculateBranchEnds3D(const point3D* p1,const
point3D* p2, float branch_angle,
              float lambda, point3D*
reduced_point, point3D* rotated_point){

        point3D pseudo_origin;
        point3D pt1_wrt_pseudo_origin;
        point3D rotated_pt_wrt_pseudo_origin;
        point3D rotated_pt_wrt_normal_origin;

        //Create Branch
        branchReduction3D(p1, p2, lambda,
&pseudo_origin);
        preProcessing3D(p2, &pseudo_origin,
&pt1_wrt_pseudo_origin);
        Rotation3D(&pt1_wrt_pseudo_origin,
branch_angle, &rotated_pt_wrt_pseudo_origin);
        postProcessing3D(&rotated_pt_wrt_pseudo_or
igin,  &pseudo_origin,
&rotated_pt_wrt_normal_origin);

        //child branch origin = parent branch head
        reduced_point->x = pseudo_origin.x;
        reduced_point->y = pseudo_origin.y;
        reduced_point->z = pseudo_origin.z;
        rotated_point->x =
rotated_pt_wrt_normal_origin.x;
        rotated_point->y =
rotated_pt_wrt_normal_origin.y;
        rotated_point->z =
rotated_pt_wrt_normal_origin.z;
}

void branchReduction3D(point3D* p0, point3D* p1,
float lambda, point3D* temp ){
        temp->x = p0->x + lambda * (p1->x - p0-
>x);
        temp->y = p0->y + lambda * (p1->y - p0-
>y);
        temp->z = p0->z + lambda * (p1->z - p0-
>z);
}
point3D branchExtension3D(point3D* p0, point3D*
p1, float lambda){
        point3D output;
        lambda = 1;
        output.x = p1->x + (lambda) * (p1->x - p0-
>x);
        output.y = p1->y + (lambda) * (p1->y - p0-
>y);
        output.z = p1->z + (lambda) * (p1->z - p0-
>z);
        return output;
}
void preProcessing3D(const point3D* p0, const
point3D* delta, point3D* output){
        output->x = p0->x - delta->x;
        output->y = p0->y - delta->y;
        output->z = p0->z - delta->z;

}
void Rotation3D(const point3D* p0, float alpha,
point3D* output){
        output->x = cos(alpha) * p0->x -
sin(alpha) * p0->y;
        output->y = sin(alpha) * p0->x +
cos(alpha) * p0->y;
        output->z = p0->z; // TODO
```

```
}
void postProcessing3D(const point3D* p0,const
point3D* delta, point3D* output){
        output->x = p0->x + delta->x;
        output->y = p0->y + delta->y;
        output->z = p0->z + delta->z;

}
```