CMPE240 Project-2 Report

Ezgi Kaya - 016042871
Computer Engineering Department, College of Engineering
San Jose State University, San Jose, CA 94303
E-mail: ezgi.kaya@sjsu.edu

Abstract

In this project, a 3D graphics processing engine is designed using an LPC11C24 CPU module. A solid cube and half of a sphere is displayed on an LCD display, from the point of view of a camera. The main challenge was to implement this computationally heavy design on a hardware with very limited flash memory and RAM. This challenge was overcome by manually optimizing the code, as well as using compiler optimization. As a result, I was able to complete the given tasks successfully.

1. Introduction

In this project, an LPC11C24 CPU module, which is a 32-bit ARM Cortex-M0 microcontroller [1], is used. The goal of this project is to display a solid cube and half of a sphere on an LCD display, whose design details are provided in the following:

Cube design:

- Two of the visible sides of the cube are decorated with the "rotating squares" and "tree" patterns that were designed in Project-1.
- The 3rd visible side of the cube is painted red using diffuse reflection, and my initials are placed on this surface.
- The shadow of the cube on the xw-yw plane is computed, and displayed with dark blue.

Half sphere design:

- The visible side of the half sphere is painted green using diffuse reflection.

This project was challenging as it requires computationally heavy tasks that must be performed using limited resources, as the CPU module has 32kB flash and 8kB RAM memory. To prevent memory overflow, I had to manually optimize the code. For this purpose, I first analyzed the .map file to see which parts of the code take up most of the memory. I used dynamic memory allocation, used smaller size data types when possible, used smaller size arrays, and simplified the calculations as much as possible. Additionally, I enabled -Os type compiler optimization, which automatically optimizes the code for size. As a result, I overcame the memory problem.

2. Methodology

In this section, the details on the design objectives, and the mathematical formulation to achieve these design objectives are provided.

2.1. Design Parameters

The following is a list of the design parameters, and their default values in my design. The values are determined such that the objects/shadows fit into the display, and are properly visible.

- Camera location, (Xe, Ye, Ze) = (350,350,220)
- Point light source location, (Xs,Ys,Zs) = (0, 200,420)
- Focal length of the camera, D_focal = 100
- Cube side length, l = 100
- Cube center location, cube_center = (150, 200, 10+l/2)
- Sphere radius, R = 180
- Sphere center is given as (0,0,0)

2.2. Transformation Pipeline

Transformation pipeline is the process of transforming a point in the world coordinates into a point in the physical display coordinates. This is done in 3 steps as shown below.

World-to-Viewer transformation:

$$x_i' = -sin\theta x_i + cos\theta y_i$$
$$y_i' = -cos\phi cos\theta x_i - cos\phi sin\theta y_i + sin\phi z_i$$
$$z_i' = -sin\phi cos\theta x_i - sin\phi cos\theta y_i - cos\phi z_i + \rho$$

where

$$\rho = \sqrt{X_e^2 + Y_e^2 + Z_e^2}$$
$$sin\theta = Y_e / \sqrt{X_e^2 + Y_e^2}$$
$$cos\theta = X_e / \sqrt{X_e^2 + Y_e^2}$$
$$sin\phi = \sqrt{X_e^2 + Y_e^2} / \rho$$
$$cos\phi = Z_e / \rho$$

Perspective projection:

$$x_i'' = D_{focal} \times x_i' / z_i'$$
$$y_i'' = D_{focal} \times y_i' / z_i'$$

Virtual to Physical:

$$x_{physical} = x_{virtual} + M/2$$
$$y_{physical} = -y_{virtual} + N/2$$

where MxN is the resolution, 128x160.

2.3. Shadow Calculation

To calculate the shadow of the cube on the xw-yw plane, we first find the lambda (λ) value for each corner point of the top surface of the cube, P_i, using the following equation.

$$\lambda = -\frac{n_x x_i + n_y y_i + n_z z_i}{n_x(X_s - x_i) + n_y(Y_s - y_i) + n_z(Z_s - z_i)}$$

where n is the normal vector perpendicular to the xw-yw plane, n = (0,0,1).

After calculating the lambda values, we calculate the coordinates of the intersection points, using the following equation.

$$P_{i,\,intersection} = P_i + \lambda(P_s - P_i)$$

2.4. Diffuse Reflection

In this design, diffuse reflection is used to determine the color of a particular point on a surface, which is affected by the angle the light hits that point, the distance of the point to the light source, and the reflectivity of the surface. The following equation shows the diffuse reflection of the primitive color red.

$$I_{dr} = K_{dr}\frac{1}{||\vec{r}||^2}\frac{\vec{r}\bullet\vec{n}}{||\vec{r}||\,||\vec{n}||}$$

where $\vec{n}$ is the normal vector perpendicular to the surface at the point of interest, and $\vec{r}$ is the vector from the point light source to the point on the surface. After calculating I_d, we do the following regularization, so that the color is between 20 and 255.

$$I_d \leftarrow 20 + 235 * (I_d - I_{d,max})/(I_{d,min} - I_{d,max})$$

After calculating the diffuse reflection at the corner points, we calculate the diffuse reflection along the borders. The points on the borderline are computed using the DDA algorithm, which is explained in the next section. Finally, bilinear interpolation is applied to calculate the diffuse reflection at the intermediate points, using the following equations.

$$I_{diff,x} = \frac{I_{diff,j} - I_{diff,i}}{x_j - x_i}x - \frac{I_{diff,j} - I_{diff,i}}{x_j - x_i}x_j + I_{diff,j}$$

$$I_{diff,y} = \frac{I_{diff,j} - I_{diff,i}}{y_j - y_i}y - \frac{I_{diff,j} - I_{diff,i}}{y_j - y_i}y_j + I_{diff,j}$$

$$I_{diff} = \frac{1}{2}(I_{diff,x} + I_{diff,y})$$

2.5. DDA Algorithm

Let's assume that Fig. 1 shows some pixels on the LCD display, and we want to draw a line from (1,1) to (2,5). In this case, since the slope is higher than 1, there would be a gap between the drawn pixels. DDA algorithm solves this problem by rewriting the equation as the following. Initially, we have:

$$y_{k+1} = y_k + b$$
$$x_{k+1} = x_k + 1$$

where b in this case is greater than 1. With DDA algorithm, we write;

$$y_{k+1} = y_k + 1$$
$$x_{k+1} = \frac{1}{b} + x_k$$

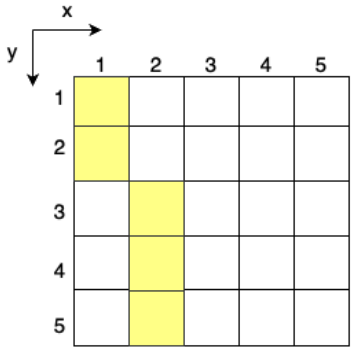

Figure 1: Visualization of the DDA algorithm.

In this example, the slope, b is 4. Following the equations given above, the locations of the pixels to be drawn in between (1,1) and (2,5) are calculated as the following.

$$k = 1: y_1 = 1, x_1 = 1$$
$$k = 2: y_2 = 2, x_2 = 1 + 0.25 = 1.25 \sim 1$$
$$k = 3: y_3 = 3, x_2 = 1.25 + 0.25 = 1.5 \sim 2$$
$$k = 4: y_4 = 4, x_2 = 1.5 + 0.25 = 1.75 \sim 2$$

2.6. Sphere Design

Fig. 2 shows the initial drawing of the half sphere during the design process. As it can be seen, the sphere consists of layers of circles drawn on top of each other. The radius of each layer is calculated according to the left image in , and is calculated as $Rsin(\gamma)$. Once we calculate the radius of a circle, we calculate the coordinates of the points on that circle, according to the right image in . The resulting

formula to calculate the x-y-z coordinates are given below. To draw the half sphere, we sweep the angels α and γ, find the x-y-z coordinates of each point, and connect those points together by using the drawline function. As a result, we obtain a sphere that consists of little patches. Finally, we calculate the diffuse reflection for each patch, and paint them with that color.

$$x = R \times sin(\gamma) \times cos(\alpha)$$
$$y = R \times sin(\gamma) \times sin(\alpha)$$
$$z = R \times cos(\gamma)$$

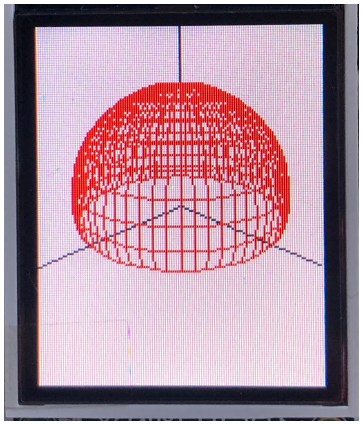

Figure 2: Initial drawing of the half sphere.

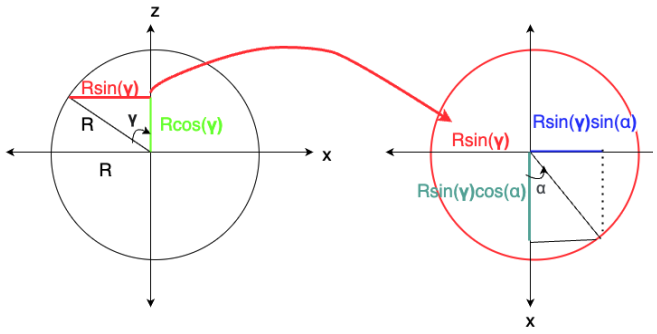

Figure 3: Sphere design method.

3. Implementation

In this section, the details on the hardware and software design are provided.

3.1. Hardware Design

The complete list of the hardware used in this project are as follows.

1. NXP LPC11C24 Eval Board [2]
2. CTIONE BoardB for embedded graphics [3]
3. TFT 128X160 1.8" LCD Display [4]
4. Female and male pin headers
5. USB C to Micro USB cable

Fig. 4 shows a simple block diagram of the entire system. The CPU module is connected to the prefabricated board, CTIONE BoardB, which is connected to the LCD display, as shown in Fig. 5. The communication between the CPU module and LCD display is carried out by the SPI interface. Fig. 6 and Fig. 7 show the pin connectivity table and the schematic of the interface of the CPU module to the LCD display, respectively.

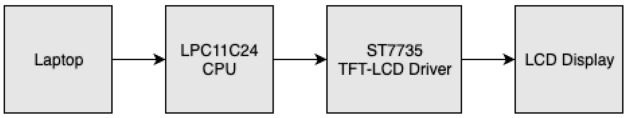

Figure 4: System block diagram.

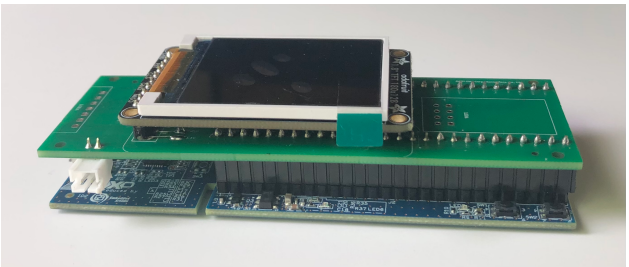

Figure 5: The prototype system.

Description	LPC11C24 pin name	LPC11C24 header name	LCD display pin name
GND	-	J2-54	GND
3.3VOUT	-	J2-28	VCC
RST	PIO2_2	J2-14	RESET
D/C	PIO2_1	J2-13	D/C
N/A	N/A	N/A	CARD_CS
SSEL0	PIO0_2	J2-8	TFT_CS
MOSI0	PIO0_9	J2-5	MOSI
SCK0	PIO2_11	J2-7	SCK
MISO0	PIO0_8	J2-6	MISO
3.3VOUT	-	J2-28	LITE

Figure 6: Pin connectivity table.

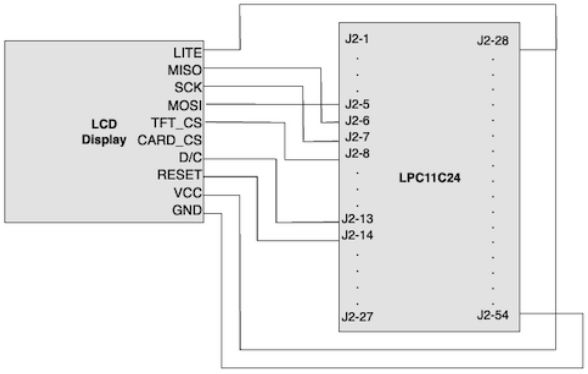

Figure 7: Schematic of the LPC11C24 interface to LCD Display.

3.2. Software Design

In this section, the flow chart of the software design and the pseudo code are provided.

3.2.1. Flow Charts

Fig. 8 shows a high level representation of the software design. After the initialization, each element is drawn and decorated/painted in an order such that the element that is closer to the camera is drawn last. and show a detailed, lower level implementation of the sphere and cube design, respectively.

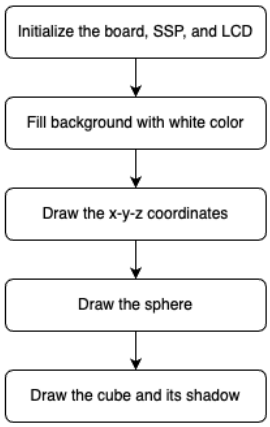

Figure 8: High level flow chart of the software design.

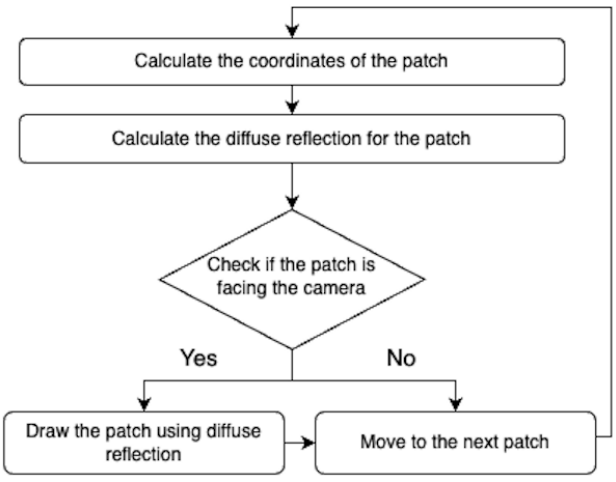

Figure 9: Zooming into the task: "draw the sphere".

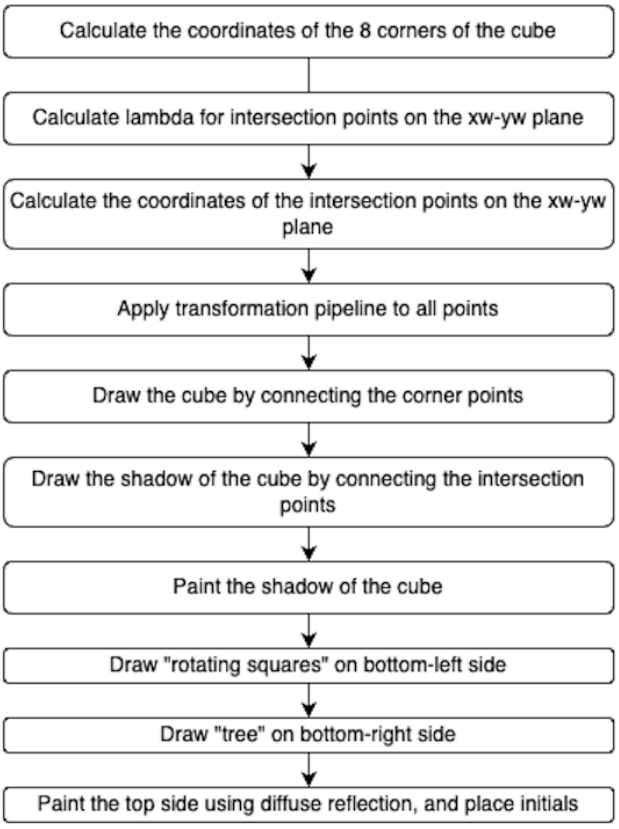

Figure 10: Zooming into the task: "draw the cube and its shadow".

3.2.2. Pseudocode

The pseudocode is provided in Fig. 11. For more details, please see the source code in the Appendix at the end of this paper.

```
initialize() // initialize global variables
board_init() // initialize the board
ssp0_init() // initialize SSP
lcd_init() // initialize the LCD display
fillrect(...) // fill background with white color
draw_coordinates() // draw x-y-z coordinates
draw_sphere():
    for each patch:
        Calculate the coordinates
        calc_normal_vecor(...) // calculate the normal vector of the patch
        calc_diffuse_reflection(...) // calculate diffuse reflection
        If the patch is facing the camera:
            transform(...) // Apply transformation pipeline
            draw the patch
draw_cube():
    Calculate the coordinates of the corners
    Calculate lambda for intersection points
    Calculate the coordinates of intersection points
    transform(...) // apply transformation pipeline
    Draw the cube borders
    Draw the shadow of the cube
    paint_rectangle(...) // paint the shadow of the cube
    rotating_squares(...) // draw “rotating squares” on bottom left side
    forest(...) // draw “tree” on bottom right side
    paint_rectangle(...) // paint the top side using diffuse reflection
    write_initials(...) // place initials on the top surface
```

Figure 11: Pseudocode.

4. Testing and Verification

In this project, I used the MCUXpresso IDE. To test and verify the code, below steps should be followed:

1. Install and open MCUXpresso IDE.
2. Connect the LPC11C24 prototype system to the laptop via micro usb cable.
3. Import the project zip file.
4. Click on “Build”
5. If a memory overflow error occurs:
    a. Right-click on the project folder
    b. Click on “Properties”
    c. Extend “C/C++ Build”
    d. Click on “Settings”
    e. Extend “MCU C Compiler”
    f. Click on “Optimization”
    g. Change optimization level -O0 to -Os, as shown in Fig. 12.
6. Finally, click on “Debug” to run the program.
7. shows a photo of the final result. One can easily change the location of the point light source (Xs,Ys,Zs) and/or the camera (Xe,Ye,Ze), as shown in Fig. 14, to see how the images change accordingly. shows an example view after changing the point light source location to (300,-200,450), and the camera location to (120,350,200).

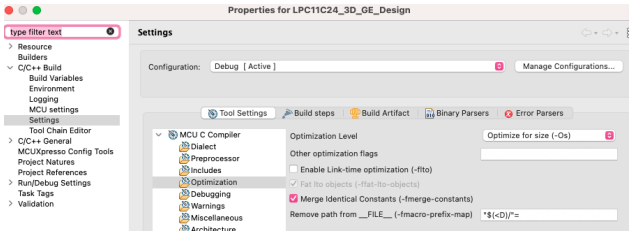

Figure 12: How to enable compiler optimization.

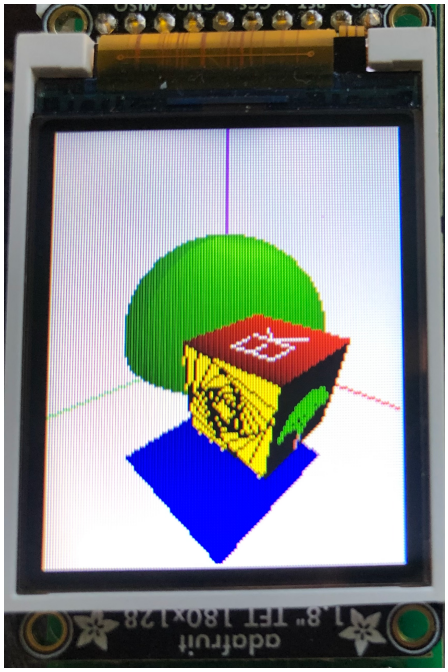

Figure 13: Final design.

```
29
30  void initialize() {
31      Xe = 350.0f, Ye = 350.0f, Ze = 220.0f;  //virtual camera location
32      Xs = 0.0; Ys = 200.0; Zs =  420.0; // point light source location
33      Rho = sqrt(pow(Xe,2) + pow(Ye,2) + pow(Ze,2));
34      D_focal = 100.0f;
```

Figure 14: How to change the point light source and camera location.

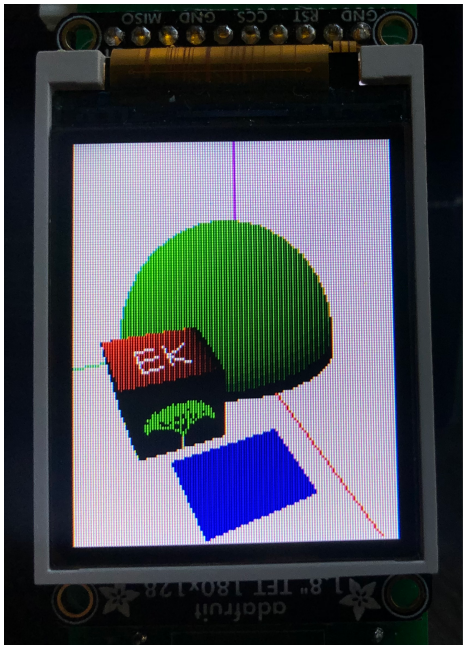

Figure 15: Final design after changing the location of the point light source and the camera.

5. Conclusion

In this project, a 3D graphics engine is designed, and a solid cube and half of a sphere is displayed on an LCD display. All the design objectives were achieved successfully. Working on this project has been very useful for me, as I got to experience performing a computationally heavy task using limited hardware resources. As future work, we can try increasing the execution speed even further. Additionally, we can modify the code such that the user can change the location of the point light source and/or the camera on the fly during run time, and observe the changes in the images.

6. Acknowledgement

I would like to thank our Professor Harry Li for his guidance on this project.

7. References

[1] LPC11Cx2/Cx4 Datasheet, https://www.nxp.com/docs/en/data-sheet/LPC11CX2_CX4.pdf (accessed Apr.23, 2023)

[2] LPCXpresso board for LPC11C24 with CMSIS DAP probe, https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc1100-arm-cortex-m0-plus-m0/lpcxpresso-board-for-lpc11c24-with-cmsis-dap-probe:OM13093 (accessed Apr.23, 2023)

[3] CTIONE BoardB for embedded graphics, https://www.ebay.com/itm/175641598709 (accessed Apr.23, 2023)

[4] 358, https://www.digikey.com/en/products/detail/adafruit-industries-llc/358/5801368 (accessed Apr.23, 2023)

8. Appendix

```
#include "lcd__api.h"
#include "board.h"
#include <math.h>
#include <stdio.h>
//#include <cr_section_macros.h>
#define UpperBD 20
#define PI 3.141592654
#define M 128
#define N 160
float Xe, Ye, Ze, Xs, Ys, Zs, Rho, D_focal, sin_theta, cos_theta, sin_phi, cos_phi;
float Idmin = 255, Idmax = -1e35;
typedef struct point
{
	float x,y,z;
};
typedef struct branch
{
	struct point base, root, left, mid, right;
};
void initialize() {
	Xe = 350.0f, Ye = 350.0f, Ze = 220.0f;  //virtual camera location
	Xs = 0.0; Ys = 200.0; Zs =  420.0; // point light source location
	Rho = sqrt(pow(Xe,2) + pow(Ye,2) + pow(Ze,2));
	D_focal = 100.0f;
	//sin and cosine computation for world-to-viewer
	sin_theta = Ye / sqrt(pow(Xe,2) + pow(Ye,2));
	cos_theta = Xe / sqrt(pow(Xe,2) + pow(Ye,2));
	sin_phi = sqrt(pow(Xe,2) + pow(Ye,2)) / Rho;
	cos_phi = Ze / Rho;
}
// Transformation pipeline
struct point transform(struct point *t)
{
	float viewerx=0.0, viewery=0.0, viewerz=0.0, perspectivex=0.0, perspectivey=0.0;
	struct point p;
	/////////////// world to viewer ///////////////
	viewerx = -sin_theta * (t->x) + cos_theta * (t->y);
	viewery = -cos_theta * cos_phi * (t->x) - cos_phi * sin_theta * (t->y) + sin_phi * (t->z);
	viewerz = -sin_phi * cos_theta * (t->x) - sin_phi * cos_theta * (t->y) - cos_phi * (t->z) + Rho;
	/////////////// perspective projection ///////////////
	perspectivex = D_focal * viewerx / viewerz ;
	perspectivey = D_focal * viewery / viewerz ;
	/////////////// virtual to physical ///////////////
	p.x = perspectivex + M/2;
	p.y = - perspectivey + N/2;
	p.z = t->z;
	return p;
}
struct point calc_normal_vector(struct point *q, struct point *p, struct point *s)
{
	struct point QP, QS, n;
	// find Q-R vector
	QP.x = p->x - q->x; QP.y = p->y - q->y; QP.z = p->z - q->z;
	// find Q-S vector
	QS.x = s->x - q->x; QS.y = s->y - q->y; QS.z = s->z - q->z;
	// find normal vector to the plane
	n.x = QP.y * QS.z - QP.z * QS.y;
	n.y = - (QP.x * QS.z - QP.z * QS.x);
	n.z = QP.x * QS.y - QP.y * QS.x;
	// normalize the vector
	n.x = n.x / sqrt(pow(n.x,2) + pow(n.y,2) + pow(n.z,2));
	n.y = n.y / sqrt(pow(n.x,2) + pow(n.y,2) + pow(n.z,2));
	n.z = n.z / sqrt(pow(n.x,2) + pow(n.y,2) + pow(n.z,2));
	return n;
}
int calc_diffuse_reflection(struct point *n, struct point *p)
{
	float Id; // Diffuse reflection
	struct point r;
	// find vector from point light source to the point of interest, p
	r.x = p->x - Xs; r.y = p->y - Ys; r.z = p->z - Zs;
	// calculate diffuse reflection
	Id = 0.8 * (r.x*n->x + r.y*n->y + r.z*n->z) / ((pow(r.x, 2) + pow(r.y,2) + pow(r.z,2)) * sqrt(pow(r.x,2) + pow(r.y, 2) + pow(r.z,2)) * sqrt(pow(n->x,2) + pow(n->y, 2) + pow(n->z,2)));
	if(Id < Idmin) Idmin = Id;
	else if(Id > Idmax) Idmax = Id;
	// regulate and return Id
	return (int)(20 + 235*(Id - Idmax)/(Idmin - Idmax));
}
```

```
void paint_rectangle(int xmin, int xmax, int ymin, int ymax, int zmin, int zmax, uint32_t color, int diffuse_reflection_on)
{
	// calculate normal vector
	struct point n;
	n.x = 0; n.y = 0; n.z = 1;
	struct point p;
	int Id;
	Idmin = 255, Idmax = -1e35; // initialize Idmin and Idmax
	for(int z=zmin; z<zmax; z++)
	{
		for(int y=ymin; y<ymax; y++)
		{
			for(int x=xmin; x<xmax; x++)
			{
				p.x = x, p.y = y; p.z = z;
				if(diffuse_reflection_on == 1)
				{
					Id = calc_diffuse_reflection(&n, &p);
					p = transform(&p);
					drawPixel(p.x, p.y, Id<<16);
				}
				else
				{
					p = transform(&p);
					drawPixel(p.x, p.y, color);
				}
			}
		}
	}
}
void rotating_squares(struct point *p1, struct point *p2, struct point *p3, struct point *p4)
{
	int num_levels = 10;
	float lambda = 0.8;
	struct point **square = (struct point **) malloc(num_levels * sizeof(struct point *));
	for (int i = 0; i < 10; i++) {
		square[i] = (struct point *) malloc(4 * sizeof(struct point));
	}
	// initial square
	square[0][0] = *p1;
	square[0][1] = *p2;
	square[0][2] = *p3;
	square[0][3] = *p4;
	for(int j=0; j<num_levels; j++)
	{
		// create the 4 points
		for(int i=0; i<4; i++)
		{
			square[j+1][i].x = square[j][i].x ;
			square[j+1][i].y = square[j][i].y + lambda * (square[j][(i+1)%4].y - square[j][i].y);
			square[j+1][i].z = square[j][i].z + lambda * (square[j][(i+1)%4].z - square[j][i].z);
		}
	}
	for(int j=0; j<num_levels; j++)
	{
		for(int i=0; i<4; i++)
		{
			// apply transformation pipeline
			square[j][i] = transform(&square[j][i]);
		}
	}
	for(int j=0; j<num_levels; j++)
	{
		for(int i=0; i<4; i++)
		{
			// connect the 4 points together
			drawLine(square[j][i].x, square[j][i].y , square[j][(i+1)%4].x, square[j][(i+1)%4].y, BLACK);
		}
	}
	free(square);
}
void draw_single_branch(struct point *base, struct point *root, struct point *left, struct point *mid, struct point *right)
{
	float alpha;
	float lambda = 0.8;
	struct point p_mid, p_root, p_right, p_left;
	mid->x = root->x + lambda * (root->x - base->x);
	mid->y = root->y;
	mid->z = root->z + lambda * (root->z - base->z);
	alpha = (330 * PI)/180;
	right->x = cos(alpha)*mid->x - sin(alpha)*mid->z + cos(alpha)*(-root->x) - sin(alpha)*(-root->z) - (-root->x);
	right->y = root->y;
	right->z = sin(alpha)*mid->x + cos(alpha)*mid->z + sin(alpha)*(-root->x) + cos(alpha)*(-root->z) - (-root->z);
	alpha = (30 * PI)/180;
	left->x = cos(alpha)*mid->x - sin(alpha)*mid->z + cos(alpha)*(-root->x) - sin(alpha)*(-root->z) - (-root->x);
	left->y = root->y;
	left->z = sin(alpha)*mid->x + cos(alpha)*mid->z + sin(alpha)*(-root->x) + cos(alpha)*(-root->z) - (-root->z);
	// apply transformation pipeline
	p_root = transform(root);
	p_mid = transform(mid);
	p_left = transform(left);
	p_right = transform(right);
	drawLine(p_mid.x, p_mid.y, p_root.x, p_root.y, DARKGREEN);
	drawLine(p_right.x, p_right.y, p_root.x, p_root.y, DARKGREEN);
	drawLine(p_left.x, p_left.y, p_root.x, p_root.y, DARKGREEN);
}
void draw_three_branches(struct branch *init_branch, struct branch *left_branch, struct branch *mid_branch, struct branch *right_branch){
	left_branch->base = init_branch->root;
	left_branch->root = init_branch->left;
	draw_single_branch(& left_branch->base, & left_branch->root, & left_branch->left, & left_branch->mid, & left_branch->right);
	mid_branch->base = init_branch->root;
	mid_branch->root = init_branch->mid;
	draw_single_branch(& mid_branch->base, & mid_branch->root, & mid_branch->left, & mid_branch->mid, & mid_branch->right);
```

```
		right_branch->base = init_branch->root;
		right_branch->root = init_branch->right;
		draw_single_branch(& right_branch->base, & right_branch->root, & right_branch->left, & right_branch->mid, & right_branch->right);
}
void forest(float x, float y, float z)
{
		int length = 20;
		int num_levels = 4;
		struct point base, root, left, mid, right;
		struct branch init_branch;
		struct point p_root, p_base;
		for(int p=0; p<1; p++)
		{
			// initial trunk
			base.x = x;
			base.y = y;
			base.z = z;
			root.x = x;
			root.y = y;
			root.z = z + length;
			// apply transformation pipeline
			p_base = transform(&base);
			p_root = transform(&root);
			drawLine(p_base.x, p_base.y, p_root.x, p_root.y, BROWN);
			draw_single_branch(&base, &root, &left, &mid, &right);
			init_branch.base = base;
			init_branch.root = root;
			init_branch.right = right;
			init_branch.mid = mid;
			init_branch.left = left;
			struct branch **arr;
			arr = (struct branch **) malloc(num_levels * sizeof(struct branch *));
			if(arr == NULL)
			{
				printf("malloc error");
			}
			arr[0] = (struct branch *) malloc(1 * sizeof(struct branch));
			if(arr[0] == NULL)
			{
				printf("malloc error");
			}
			arr[0][0] = init_branch;
			for(int level=1; level<num_levels; level++)
			{
				arr[level] = (struct branch *) malloc((int)pow(3, level) * sizeof(struct branch));
				if(arr[level] == NULL)
				{
					printf("malloc error");
				}
				for(int i = 0, j = 0; i<pow(3, level-1) && j < pow(3, level); i++, j+=3)
				{

draw_three_branches(&arr[level-1][i], &arr[level][j], &arr[level][j+1], &arr[level][j+2]);
				}
				free(arr[level-1]);
			}
			free(arr[num_levels-1]);
			free(arr);
		}
}
void write_initials(float x_init, float y_init, float z_init)
{
		struct point* letter = malloc(14 * sizeof(struct point));
		float inc = 20;
		int i = 0;
		letter[i].x = x_init; letter[i].y = y_init; letter[i].z = z_init; i++;
		letter[i].x = x_init + inc; letter[i].y = y_init + inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + inc; letter[i].y = y_init + inc; letter[i].z = z_init; i++;
		letter[i].x = x_init; letter[i].y = y_init + 2*inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + inc; letter[i].y = y_init; letter[i].z = z_init; i++;
		letter[i].x = x_init + inc; letter[i].y = y_init + 2*inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + 1.5*inc; letter[i].y = y_init; letter[i].z = z_init; i++;
		letter[i].x = x_init + 2.5*inc; letter[i].y = y_init; letter[i].z = z_init; i++;
		letter[i].x = x_init + 1.5*inc; letter[i].y = y_init + inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + 2.5*inc; letter[i].y = y_init + inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + 1.5*inc; letter[i].y = y_init + 2*inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + 2.5*inc; letter[i].y = y_init + 2*inc; letter[i].z = z_init; i++;
		letter[i].x = x_init + 2.5*inc; letter[i].y = y_init; letter[i].z = z_init; i++;
		letter[i].x = x_init + 2.5*inc; letter[i].y = y_init + 2*inc; letter[i].z = z_init;
		for(int n = 0; n <= i; n++)
		{
			// apply transformation pipeline
			letter[n] = transform(&letter[n]);
		}
		for(int n = 0; n < i; n++)
		{
			if(n%2 == 1) n++;
			drawLine(letter[n].x, letter[n].y, letter[n+1].x, letter[n+1].y, WHITE);
		}
		free(letter);
}
void draw_coordinates()
{
		struct point* coor = malloc(4 * sizeof(struct point));
		// Coordinates
		coor[0].x = 0.0; coor[0].y = 0.0; coor[0].z = 0.0; // origin
		coor[1].x = 400.0; coor[1].y = 0.0; coor[1].z = 0.0; // x-axis
		coor[2].x = 0.0; coor[2].y = 400.0; coor[2].z = 0.0; // y-axis
		coor[3].x = 0.0; coor[3].y = 0.0; coor[3].z = 400.0; // z-axis
		// apply transformation pipeline
		for(int i = 0; i <= 3; i++)
		{
			coor[i] = transform(&coor[i]);
		}
		// Draw coordinates
```

```
		drawLine(coor[0].x, coor[0].y, coor[1].x, coor[1].y, MAGENTA);
		drawLine(coor[0].x, coor[0].y, coor[2].x, coor[2].y, RED);
		drawLine(coor[0].x, coor[0].y, coor[3].x, coor[3].y, PURPLE);
		free(coor);
}
void draw_cube()
{
		struct point* world = malloc(15 * sizeof(struct point));
		struct point* physical = malloc(15 * sizeof(struct point));
		float l = 100.0f; // cube side length
		struct point cube_center;
		cube_center.x = 150; cube_center.y = 200; cube_center.z = 10 + l/2;
		// Cube
		world[0].x = cube_center.x + l/2; world[0].y = cube_center.y + l/2; world[0].z = cube_center.z + l/2; // P_0
		world[1].x = cube_center.x - l/2; world[1].y = cube_center.y + l/2; world[1].z = cube_center.z + l/2; // P_1
		world[2].x = cube_center.x - l/2; world[2].y = cube_center.y - l/2; world[2].z = cube_center.z + l/2; // P_2
		world[3].x = cube_center.x + l/2; world[3].y = cube_center.y - l/2; world[3].z = cube_center.z + l/2; // P_3
		world[4].x = cube_center.x + l/2; world[4].y = cube_center.y + l/2; world[4].z = cube_center.z - l/2; // P_4
		world[5].x = cube_center.x - l/2; world[5].y = cube_center.y + l/2; world[5].z = cube_center.z - l/2; // P_5
		world[6].x = cube_center.x - l/2; world[6].y = cube_center.y - l/2; world[6].z = cube_center.z - l/2; // P_6
		world[7].x = cube_center.x + l/2; world[7].y = cube_center.y - l/2; world[7].z = cube_center.z - l/2; // P_7
		world[8].x = Xs; world[8].y = Ys; world[8].z = Zs; // P_S (point light source)
		world[9].x = 0.0; world[9].y = 0.0; world[9].z = 0.0; // arbitrary vector a on xw-yw plane
		world[10].x = 0.0; world[10].y = 0.0; world[10].z = 1.0; // normal vector n for xw-yw plane
		// lambda for Intersection points on xw-yw plane
		float lambda[4];
		lambda[0] = - (world[10].x * world[0].x + world[10].y * world[0].y + world[10].z * world[0].z) /
				(world[10].x * (world[8].x - world[0].x) + world[10].y * (world[8].y - world[0].y) + world[10].z * (world[8].z - world[0].z));
		lambda[1] = - (world[10].x * world[1].x + world[10].y * world[1].y + world[10].z * world[1].z) /
				(world[10].x * (world[8].x - world[1].x) + world[10].y * (world[8].y - world[1].y) + world[10].z * (world[8].z - world[1].z));
		lambda[2] = - (world[10].x * world[2].x + world[10].y * world[2].y + world[10].z * world[2].z) /
				(world[10].x * (world[8].x - world[2].x) + world[10].y * (world[8].y - world[2].y) + world[10].z * (world[8].z - world[2].z));
		lambda[3] = - (world[10].x * world[3].x + world[10].y * world[3].y + world[10].z * world[3].z) /
				(world[10].x * (world[8].x - world[3].x) + world[10].y * (world[8].y - world[3].y) + world[10].z * (world[8].z - world[3].z));
		// intersection points on the xw-yw plane
		world[11].x = world[0].x + lambda[0]*(world[8].x - world[0].x); world[11].y = world[0].y + lambda[0]*(world[8].y - world[0].y); world[11].z = 0.0; // intersection point for P_0
		world[12].x = world[1].x + lambda[1]*(world[8].x - world[1].x); world[12].y = world[1].y + lambda[1]*(world[8].y - world[1].y); world[12].z = 0.0; // intersection point for P_1
		world[13].x = world[2].x + lambda[2]*(world[8].x - world[2].x); world[13].y = world[2].y + lambda[2]*(world[8].y - world[2].y); world[13].z = 0.0; // intersection point for P_2
		world[14].x = world[3].x + lambda[3]*(world[8].x - world[3].x); world[14].y = world[3].y + lambda[3]*(world[8].y - world[3].y); world[14].z = 0.0; // intersection point for P_3
		// apply transformation pipeline
		for(int i = 0; i <= 14; i++)
		{
			physical[i] = transform(&world[i]);
		}
		// Draw the cube
		drawLine(physical[0].x, physical[0].y, physical[1].x, physical[1].y, BLACK);
		drawLine(physical[1].x, physical[1].y, physical[2].x, physical[2].y, BLACK);
		drawLine(physical[2].x, physical[2].y, physical[3].x, physical[3].y, BLACK);
		drawLine(physical[3].x, physical[3].y, physical[0].x, physical[0].y, BLACK);
		drawLine(physical[4].x, physical[4].y, physical[5].x, physical[5].y, BLACK);
		drawLine(physical[7].x, physical[7].y, physical[4].x, physical[4].y, BLACK);
		drawLine(physical[0].x, physical[0].y, physical[4].x, physical[4].y, BLACK);
		drawLine(physical[1].x, physical[1].y, physical[5].x, physical[5].y, BLACK);
		drawLine(physical[3].x, physical[3].y, physical[7].x, physical[7].y, BLACK);
//		// Draw rays from point light source to cube
//		drawLine(physical[8].x, physical[8].y, physical[11].x, physical[11].y, YELLOW);
//		drawLine(physical[8].x, physical[8].y, physical[12].x, physical[12].y, YELLOW);
//		drawLine(physical[8].x, physical[8].y, physical[13].x, physical[13].y, YELLOW);
//		drawLine(physical[8].x, physical[8].y, physical[14].x, physical[14].y, YELLOW);
		// Draw the shadow of the cube
		drawLine(physical[11].x, physical[11].y, physical[12].x, physical[12].y, DARKBLUE);
		drawLine(physical[12].x, physical[12].y, physical[13].x, physical[13].y, DARKBLUE);
		drawLine(physical[13].x, physical[13].y, physical[14].x, physical[14].y, DARKBLUE);
		drawLine(physical[14].x, physical[14].y, physical[11].x, physical[11].y, DARKBLUE);
		// Paint the shadow of the cube
		paint_rectangle(world[13].x, world[14].x, world[14].y, world[11].y, world[11].z, world[11].z + 1, DARKBLUE, 0); // background
		// Draw the rotating squares on bottom-left side of the cube
		paint_rectangle(world[4].x, world[4].x + 1, world[7].y, world[4].y, world[4].z, world[0].z, YELLOW, 0); // background
		rotating_squares(&world[0], &world[3], &world[7], &world[4]);
		// Draw a tree on bottom-right side of the cube
		paint_rectangle(world[5].x, world[4].x, world[4].y, world[4].y + 1, world[4].z, world[0].z, BLACK, 0); // background
		forest((world[5].x + world[4].x)/2, world[4].y, world[4].z);
		// paint the top surface of the cube, with diffuse reflection
```

```
		paint_rectangle((int)world[2].x, (int)world[0].x, (int)world[2].y, (int)world[0].y, (int)world[0].z, (int)world[0].z + 1, RED, 1);
		// write my initials "EK" on the cube
		write_initials(world[2].x + l/4 , world[2].y + l/3, world[2].z);
		free(world);
		free(physical);
}
void draw_sphere()
{
		float alpha = 0.0f;
		float gamma = 90.0f;
		float R = 180; // radius of the sphere
		int m = 90, n = 360;
		struct point* sphere = malloc(m * sizeof(struct point));
		struct point normal;
		struct point cam_to_patch;
		struct point patch_to_cam;
		int Id;
		Idmin = 255, Idmax = -1e35; // initialize Idmin and Idmax
		for(int j = 0; j < n + 1; j++)
		{
			gamma = 90.0f;
			for(int i = 0; i < m; i++)
			{
				sphere[i].x = R*sin(gamma * PI/180) * cos(alpha * PI/180);
				sphere[i].y = R*sin(gamma * PI/180) * sin(alpha * PI/180);
				sphere[i].z = R*cos(gamma * PI/180);
				// calculate vector from camera to patch
				cam_to_patch.x = Xe - sphere[i].x;
				cam_to_patch.y = Ye - sphere[i].y;
				cam_to_patch.z = 0;
				// calculate vector from patch to camera
				patch_to_cam.x = sphere[i].x;
				patch_to_cam.y = sphere[i].y;
				patch_to_cam.z = 0;
				// calculate diffuse reflection for the current patch
				Id = calc_diffuse_reflection(&sphere[i], &sphere[i]);
				// draw the patch, if it is facing the camera
				if(patch_to_cam.x * cam_to_patch.x + patch_to_cam.y * cam_to_patch.y + patch_to_cam.z * cam_to_patch.z > 0)
				{

drawPixel(transform(&sphere[i]).x, transform(&sphere[i]).y, Id<<8);
				}
				gamma -= 90/m;
			}
			alpha += 360/n;
		}
		free(sphere);
}
int main(void)
{
		initialize();
		board_init();
		ssp0_init();
		lcd_init();
		fillrect(0, 0, ST7735_TFTWIDTH, ST7735_TFTHEIGHT, WHITE);
		draw_coordinates();
		draw_sphere();
		draw_cube();
		return 0;
}
```