UCS505

Computer Graphics

Project Report On

TOWER OF HANOI PROBLEM

Submitted by: -

Chirag Mohan Gupta (102103554)

Aditya Singh (102103558)

Himanshu Bansal (102103568)

Group: 3COE 20

Submitted to: -

Dr. Anupam Garg

COMPUTER ENGINEERING DEPARTMENT

THAPAR INSTITUTE OF ENGINEERING AND TECHNOLOGY,

PATIALA, PUNJAB,

INDIA

Jan-May 2024

TABLE OF CONTENTS

INTRODUCTION

The **Tower of Hanoi** project is a simulation developed using OpenGL and GLUT in C/C++. It provides an interactive environment where users can solve the Tower of Hanoi puzzle.

The Tower of Hanoi is a classic mathematical puzzle that involves three rods and disks of varying sizes. The objective is to move the entire stack of disks from one rod to another, following specific rules. Initially, the disks are stacked on one rod in decreasing order of size, resembling a cone with the smallest disk on top. The rules for the Tower of Hanoi puzzle are as follows:

1. Only one disk can be moved at a time.
2. Each move consists of taking the uppermost disk from one stack and placing it on top of another stack or an empty rod.
3. A disk cannot be placed on top of a smaller disk.

This puzzle can be solved in a minimum number of moves, which is **2^n - 1**, where n is the number of disks. The solution involves recursive steps where disks are moved between the rods systematically until the entire stack is transferred to a different rod while adhering to the rules mentioned above.

The key features of this project include:

1. **Interactive Environment:** Users can interact with the Tower of Hanoi puzzle through keyboard inputs to solve the puzzle.
2. **Realistic Graphics:** The project utilizes OpenGL to render realistic 3D graphics of the rods and disks, providing a visually appealing experience.
3. **Dynamic Animation:** The disks can be moved between rods, and the simulation includes animations to illustrate the movement of the disks.
4. **User Controls:** Users can control the speed of the animation and solve the puzzle from the initial state using keyboard inputs.
5. **Educational Purpose:** The project serves an educational purpose by allowing users to learn about the Tower of Hanoi puzzle and its solution algorithm.

WORKING

Initialization: The program initializes the game environment, including the rods, discs, and active disc state. It sets up the OpenGL display and keyboard callbacks.

Display: The display_handler () function is responsible for rendering the game environment, including the base, rods, and discs.

Keyboard Controls:

- Pressing 'ESC', 'q', or 'Q' quits the program.
- Pressing 's' or 'S' solves the Tower of Hanoi puzzle from the initial state.
- Pressing '+' increases the animation speed.
- Pressing '-' decreases the animation speed.

Animation: The anim_handler () function handles the animation of moving discs between rods. It updates the positions and orientations of discs based on their current state.

Solving: The Tower of Hanoi puzzle can be solved automatically by pressing 's' or 'S'. The program uses a recursive algorithm to find the optimal solution and animates the movements accordingly.

Interpolation: Disc movement is interpolated using Hermite interpolation to achieve smooth animation between start and destination positions.

Overall, the program provides a visual representation of the Tower of Hanoi puzzle and allows users to interact with it using keyboard controls. Users can solve the puzzle manually or let the program solve it automatically. The animation speed can also be adjusted for a better user experience.

CONCEPTS USED

This Tower of Hanoi simulation project utilizes several concepts of computer graphics to create an interactive and visually appealing representation of the puzzle. Here are some of the key concepts used:

1. **OpenGL and GLUT**: The project relies on OpenGL (Open Graphics Library) for rendering 2D and 3D graphics, and GLUT (OpenGL Utility Toolkit) for handling windowing and user input. These libraries provide a framework for creating graphical applications and managing rendering contexts.
2. **3D Modeling:** The discs and rods in the Tower of Hanoi simulation are modeled in 3D space using geometric primitives such as cylinders and tori. The positions and orientations of these objects are computed and updated dynamically to reflect the current state of the puzzle.
3. **Transformation:** Transformation matrices are used to translate, rotate, and scale objects in 3D space. These transformations are applied to the discs and rods to position them correctly within the game environment and to animate their movements between rods.
4. **Lighting and Shading:** Lighting models are used to simulate the interaction of light with surfaces, creating realistic lighting effects such as diffuse and specular reflections. Shading techniques, such as smooth shading, are employed to calculate the color of pixels on surfaces based on their orientation relative to light sources.
5. **Hermite Interpolation:** Hermite interpolation is used to interpolate between keyframe positions of moving discs, creating smooth animation transitions. This technique ensures that the movement of discs appears fluid and natural as they slide between rods.
6. **Event Handling**: User input events, such as keyboard presses, are captured and processed to enable interaction with the simulation. Event handling allows users to control the simulation, solve the puzzle, adjust animation speed, and access help information.

Overall, by leveraging these concepts of computer graphics, the Tower of Hanoi simulation project creates an immersive and engaging experience for users, allowing them to interact with the classic puzzle in a visually appealing virtual environment.

GRAPHICS FUNCTIONS AND REQUIREMENTS

Header Files:

1. **<iostream>:** Standard input-output stream.
2. **<stdio.h>:** Standard input-output library for C language.
3. **<GL/glut.h>:** OpenGL Utility Toolkit, used for creating graphical user interfaces.
4. **<cmath>:** C++ mathematical functions.
5. **<list>:** Standard template library for lists in C++

Graphics Functions:

1. **draw_solid_cylinder(double x, double y, double r, double h):** Draws a solid cylinder.
2. **draw_board_and_rods(GameBoard const& board):** Draws the game board and rods.
3. **draw_discs():** Draws the discs on the rods.
4. **display_handler():** Displays the game graphics.
5. **reshape_handler(int w, int h):** Handles window reshaping.
6. **render(void):** Renders the initial display screen.

User Defined Functions:

1. **initialize():** Initializes OpenGL settings and game.
2. **initialize_game():** Initializes game objects and parameters.
3. **towers():** Main function to set up the game window and initialize GLUT.
4. **move_stack(int n, int f, int t):** Recursive function to move discs between rods.
5. **solve():** Solves the Tower of Hanoi problem.
6. **keyboard_handler(unsigned char key, int x, int y):** Handles keyboard inputs during the game.
7. **reshape(int w, int h):** Reshapes the window viewport.
8. **drawStrokeText(const char* string, int x, int y, int z):** Draws stroke text on the screen.
9. render(void): Renders the initial display screen.

10. **keyboard_handler_for_intro(unsigned char key, int x, int y):** Handles keyboard inputs during the initial screen.
11. **main(int argc, char* argv[]):** Main function to initialize GLUT and start the program.
12. **move_disc(int from_rod, int to_rod):** Moves a disc from one rod to another.
13. **get_inerpolated_coordinate(Vector3 sp, Vector3 tp, double u):** Calculates interpolated coordinates for animation.
14. **normalize(Vector3& v):** Normalizes a vector.
15. **operator-(Vector3 const& v1, Vector3 const& v2):** Overloaded subtraction operator for vectors.
16. **anim_handler():** Handles animation updates.

SOURCE CODE

```
#include <iostream>
#include<stdio.h>
#include <GL/glut.h>
#include <cmath>
#include <list>
#include<GL/glut.h>

#define NUM_DISCS 3
#define ROD_HEIGHT 5
#define WINDOW_WIDTH 1350
#define WINDOW_HEIGHT 690
#define PI 22/7.0f
#define DISC_SPACING 0.33
#define BOARD_X 10
#define BOARD_Y 5

using namespace std;

struct Vector3 {
	double x, y, z;
	Vector3() { x = y = z = 0.0; }
	Vector3(double x, double y, double z) : x(x), y(y), z(z) { }
	Vector3(Vector3 const& rhs) { *this = rhs; }
	Vector3& operator= (Vector3 const& rhs)
	{
		x = rhs.x;
		y = rhs.y;
		z = rhs.z;
		return *this;
	}
};
```

```
struct Disc {
	Disc() { normal = Vector3(0.0, 0.0, 1.0); }

	Vector3 position; // Location
	Vector3 normal;   // Orientation
};

struct ActiveDisc {				// Active Disc to be moved [later in motion]
	int disc_index;
	Vector3 start_pos, dest_pos;
	double u;					// u E [0, 1]
	double step_u;
	bool is_in_motion;
	int direction;			// +1 for Left to Right & -1 for Right to left, 0 =
stationary
};

struct Rod {
	Vector3 positions[NUM_DISCS];
	int occupancy_val[NUM_DISCS];
};

struct GameBoard {
	double x_min, y_min, x_max, y_max;	// Base in XY-Plane
	double rod_base_rad;			// Rod's base radius
	Rod rods[3];
};

struct solution_pair {
	size_t f, t;			// f = from, t = to
};

// Game Globals
```

```
Disc discs[NUM_DISCS];
GameBoard t_board;
ActiveDisc active_disc;
list<solution_pair> sol;
bool to_solve = false;

// Globals for window, time, FPS, moves
float SPEED = 2;
int FPS = int(30 * SPEED);
int moves = 0;
int prev_time = 0;
int window_width = WINDOW_WIDTH, window_height = WINDOW_HEIGHT;

void initialize();
void initialize_game();
void display_handler();
void reshape_handler(int w, int h);
void keyboard_handler(unsigned char key, int x, int y);
void anim_handler();
void move_disc(int from_rod, int to_rod);
Vector3 get_inerpolated_coordinate(Vector3 v1, Vector3 v2, double u);
void move_stack(int n, int f, int t);

int towers()
{
	glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
	glutInitWindowSize(window_width, window_height);
	glutInitWindowPosition(0, 0);
	glutCreateWindow("Towers of Hanoi");
	glutDestroyWindow(1);

	initialize();
	cout << "Tower of Hanoi" << endl;
	cout << "Press H for Help" << endl;
```

```
	// Callbacks
	glutDisplayFunc(display_handler);
	glutReshapeFunc(reshape_handler);
	glutKeyboardFunc(keyboard_handler);
	glutIdleFunc(anim_handler);

	glutMainLoop();
	return 0;
}

void initialize()
{
	// Setting the clear color
	glClearColor(1, 1, 1, 0);

	// SMOOTH Shading
	glShadeModel(GL_SMOOTH);

	// Enabling Depth Test
	glEnable(GL_DEPTH_TEST);

	// Setting Light0 parameters
	GLfloat light0_pos[] = { 0.0f, 0.0f, 0.0f, 1.0f };

	// A positional light
	glLightfv(GL_LIGHT0, GL_POSITION, light0_pos);

	// Enabling Lighting
	glEnable(GL_LIGHTING);

	// Enabling Light0
	glEnable(GL_LIGHT0);
```

```
	initialize_game();
}

void initialize_game()
{
	// Initializing
	// 1) GameBoard t_board
	// 2) Discs discs
	// 3) ActiveDisc active_disc State

	// 1) Initializing GameBoard
	t_board.rod_base_rad = 1.0;
	t_board.x_min = 0.0;
	t_board.x_max = BOARD_X * t_board.rod_base_rad;
	t_board.y_min = 0.0;
	t_board.y_max = BOARD_Y * t_board.rod_base_rad;

	double x_center = (t_board.x_max - t_board.x_min) / 2.0;
	double y_center = (t_board.y_max - t_board.y_min) / 2.0;

	double dx = (t_board.x_max - t_board.x_min) / 3.0;	// Since 3 rods
	double r = t_board.rod_base_rad;

	// Initializing Rods Occupancy value
	for (int i = 0; i < 3; i++)
	{
		for (int h = 0; h < NUM_DISCS; h++)
		{
			if (i == 0)
			{
				t_board.rods[i].occupancy_val[h] = NUM_DISCS - 1 - h;
			}
			else
				t_board.rods[i].occupancy_val[h] = -1;
```

```
        }
    }

    // Initializing Rod positions
    for (int i = 0; i < 3; i++)
    {
        for (int h = 0; h < NUM_DISCS; h++)
        {
            double x = x_center + ((int)i - 1) * dx;
            double y = y_center;
            double z = (h + 1) * DISC_SPACING;
            Vector3& pos_to_set = t_board.rods[i].positions[h];
            pos_to_set.x = x;
            pos_to_set.y = y;
            pos_to_set.z = z;
            printf("%f %f %f \n", x, y, z);
        }
    }

    //2) Initializing Discs
    for (size_t i = 0; i < NUM_DISCS; i++)
    {
        // Normals are initialized whie creating a Disc object - ie in constructor of Disc
        discs[i].position = t_board.rods[0].positions[NUM_DISCS - i - 1];
    }

    //3) Initializing Active Disc
    active_disc.disc_index = -1;
    active_disc.is_in_motion = false;
    active_disc.step_u = 0.015;
    active_disc.u = 0.0;
    active_disc.direction = 0;
}
```

```
// Draw function for drawing a cylinder given position and radius and height
void draw_solid_cylinder(double x, double y, double r, double h)
{
	GLUquadric* q = gluNewQuadric();
	GLint slices = 50;
	GLint stacks = 10;

	glPushMatrix();
	glTranslatef(x, y, 0.0f);
	gluCylinder(q, r, r, h, slices, stacks);
	glTranslatef(0, 0, h);
	gluDisk(q, 0, r, slices, stacks);
	glPopMatrix();
	gluDeleteQuadric(q);
}

// Draw function for drawing rods on a given game board i.e. base
void draw_board_and_rods(GameBoard const& board)
{
	// Materials,
	GLfloat mat_white[] = { 1.0f, 1.0f, 1.0f, 1.0f };
	GLfloat mat_black[] = { 0.0f, 0.0f, 0.1f, 0.5f };

	glPushMatrix();
	// Drawing the Base Rectangle [where the rods are placed]
	glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
	glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_white);
	glBegin(GL_QUADS);
	glNormal3f(0.0f, 0.0f, 1.0f);
	glVertex2f(board.x_min, board.y_min);
	glVertex2f(board.x_min, board.y_max);
	glVertex2f(board.x_max, board.y_max);
	glVertex2f(board.x_max, board.y_min);
	glEnd();
```

```
    //Drawing Rods and Pedestals
    glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, mat_black);

    double r = board.rod_base_rad;
    for (int i = 0; i < 3; i++)
    {
        Vector3 const& p = board.rods[i].positions[0];
        draw_solid_cylinder(p.x, p.y, r * 0.1, ROD_HEIGHT - 0.1);
        draw_solid_cylinder(p.x, p.y, r, 0.1);
    }

    glPopMatrix();
}

// Draw function for drawing discs
void draw_discs()
{
    int slices = 100;
    int stacks = 10;

    double rad;

    GLfloat r, g, b;
    GLfloat emission[] = { 0.4f, 0.4f, 0.4f, 1.0f };
    GLfloat no_emission[] = { 0.0f, 0.0f, 0.0f, 1.0f };
    GLfloat material[] = { 1.0f, 1.0f, 1.0f, 1.0f };
    for (size_t i = 0; i < NUM_DISCS; i++)
    {
        switch (i)
        {
        case 0: r = 1.0; g = 0.0; b = 0.0;  // Red
            break;
        case 1: r = 0.0; g = 1.0; b = 0.0;  // Green
```

```
        break;
case 2: r = 0.0; g = 0.0; b = 1.0;  // Blue
        break;
case 3: r = 1.0; g = 1.0; b = 0.0;  // Yellow
        break;
case 4: r = 0.7; g = 1.0; b = 1.0;  // Light Blue
        break;
case 5: r = 0.8; g = 0.1; b = 0.8;  // Purple
        break;
case 6: r = 0.2; g = 1.0; b = 0.8;  // Magenta
        break;
// We can add more cases for additional colors if needed

default: r = g = b = 1.0;  // Default to white
        break;
};

material[0] = r;
material[1] = g;
material[2] = b;
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE, material);

GLfloat u = 0.0f;

//This part is written to highlight the disc in motion
if (i == active_disc.disc_index)
{
        glMaterialfv(GL_FRONT, GL_EMISSION, emission);
        u = active_disc.u;
}

GLfloat factor = 1.0f;
switch (i) {
case 0: factor = 0.2;
```

```
                break;
            case 1: factor = 0.4;
                break;
            case 2: factor = 0.6;
                break;
            case 3: factor = 0.8;
                break;
            case 4: factor = 1.2;
                break;
            case 5: factor = 1.4;
                break;
            case 6: factor = 1.6;
                break;
            case 7: factor = 1.8;
                break;
            default: break;
            };

            rad = factor * t_board.rod_base_rad;
            int d = active_disc.direction;

            glPushMatrix();
            glTranslatef(discs[i].position.x, discs[i].position.y, discs[i].position.z);
            double theta = acos(discs[i].normal.z);
            theta *= 180.0f / PI;
            glRotatef(d * theta, 0.0f, 1.0f, 0.0f);
            glutSolidTorus(0.2 * t_board.rod_base_rad, rad, stacks, slices);
            glPopMatrix();
            glMaterialfv(GL_FRONT, GL_EMISSION, no_emission);
        }
}

void display_handler()
{
```

```
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    double x_center = (t_board.x_max - t_board.x_min) / 2.0;
    double y_center = (t_board.y_max - t_board.y_min) / 2.0;
    double r = t_board.rod_base_rad;

    static float view[] = { 0,0,0 };
    view[0] = x_center;
    view[1] = y_center - 10;
    view[2] = 3 * r;

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(view[0], view[1], view[2],
        x_center, y_center, 3.0,
        0.0, 0.0, 1.0);

    glPushMatrix();
    draw_board_and_rods(t_board);
    draw_discs();
    glPopMatrix();
    glFlush();
    glutSwapBuffers();
}

void reshape_handler(int w, int h)
{
    window_width = w;
    window_height = h;

    glViewport(0, 0, (GLsizei)w, (GLsizei)h);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
```

```
		gluPerspective(45.0, (GLfloat)w / (GLfloat)h, 0.1, 20.0);

		glMatrixMode(GL_MODELVIEW);
		glLoadIdentity();
}

void move_stack(int n, int f, int t)
{
		if (n == 1) {
				solution_pair s;
				s.f = f;
				s.t = t;
				sol.push_back(s);				// Pushing the (from, to) pair of
solution to a list [so that it can be animated later]
				moves++;
				cout << "From rod " << f << " to Rod " << t << endl;
				return;
		}
		move_stack(n - 1, f, 3 - t - f);
		move_stack(1, f, t);
		move_stack(n - 1, 3 - t - f, t);
}

// Solve from 1st rod to 2nd
void solve()
{
		move_stack(NUM_DISCS, 0, 2);
}

void keyboard_handler(unsigned char key, int x, int y)
{
		//Console Outputs
		switch (key)
		{
```

```
		case 27:
		case 'q':
		case 'Q':
			exit(0);
			break;

		case 'h':
		case 'H':
			cout << "ESC: Quit" << endl;
			cout << "S: Solve from Initial State" << endl;
			cout << "H: Help" << endl;
			break;

		case 's':
		case 'S':
			if (t_board.rods[0].occupancy_val[NUM_DISCS - 1] < 0)
				break;

			solve();
			to_solve = true;
			break;

		case '+': if (SPEED < 50)SPEED += 0.2; break;
		case '-': if (SPEED > 1)SPEED -= 0.2; break;

		default:
			break;
		};
}

void reshape(int w, int h)
{
		glViewport(0, 0, w, h);
		glMatrixMode(GL_PROJECTION);
```

```
	glLoadIdentity();
	gluOrtho2D(0, w, h, 0);
	glMatrixMode(GL_MODELVIEW);
	glLoadIdentity();

}
void drawStrokeText(const char* string, int x, int y, int z)
{
	const char* c;
	glPushMatrix();
	glTranslatef(x, y + 8, z);
	glScalef(0.49f, -0.508f, z);

	for (c = string; *c != '\0'; c++)
	{
		glutStrokeCharacter(GLUT_STROKE_MONO_ROMAN, *c);
	}
	glPopMatrix();
}

void render(void)
{
	glClear(GL_COLOR_BUFFER_BIT);
	glLoadIdentity();

	glColor3f(1, 1, 1);
	drawStrokeText("Tower of Hanoi Simulation", 100, 150, 0);

	drawStrokeText("Press K to continue", 200, 400, 0);

	glutSwapBuffers();
}

void keyboard_handler_for_intro(unsigned char key, int x, int y)
```

```
{
	// Console Outputs
	switch (key)
	{
	case 27:
	case 'q':
	case 'Q':
		exit(0);
		break;

	case 'h':
	case 'H':
		cout << "ESC: Quit" << endl;
		cout << "S: Solve from Initial State" << endl;
		cout << "H: Help" << endl;
		break;

	case 'k':
	case 'K':
		glutDisplayFunc(display_handler);
		glutReshapeFunc(reshape_handler);
		glutKeyboardFunc(keyboard_handler);
		glutIdleFunc(display_handler);

		towers();

	default:
		break;
	};
}

int main(int argc, char* argv[])
{
	// Initialize glut
```

```
        glutInit(&argc, argv);

        // Specify the display mode to be RGB and single buffering
        // We use single buffering since this will be non animated
        glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE);

        // Define the size
        glutInitWindowSize(1350, 690);

        // glutFullScreen();
        // The position where the window will appear
        glutInitWindowPosition(0, 0);
        glutCreateWindow("Towers Of Hanoi");
        glutKeyboardFunc(keyboard_handler_for_intro);
        glutDisplayFunc(render);

        glutReshapeFunc(reshape);

        // Enter the main loop
        glutMainLoop();
        return 0;
}

void move_disc(int from_rod, int to_rod)
{
        int d = to_rod - from_rod;

        if (d > 0)
                active_disc.direction = 1;
        else if (d < 0)
                active_disc.direction = -1;

        if ((from_rod == to_rod) || (from_rod < 0) || (to_rod < 0) || (from_rod > 2) || (to_rod >
2))
```

```
            return;

      int i;
      for (i = NUM_DISCS - 1; i >= 0 && t_board.rods[from_rod].occupancy_val[i] < 0; i-
-);
      if ((i < 0) || (i == 0 && t_board.rods[from_rod].occupancy_val[i] < 0))
            return;

      // Either the index < 0 or index at 0 and occupancy < 0 => it's an empty rod

      active_disc.start_pos = t_board.rods[from_rod].positions[i];
      active_disc.disc_index = t_board.rods[from_rod].occupancy_val[i];
      active_disc.is_in_motion = true;
      active_disc.u = 0.0;

      int j;
      for (j = 0; j < NUM_DISCS - 1 && t_board.rods[to_rod].occupancy_val[j] >= 0; j++);
      active_disc.dest_pos = t_board.rods[to_rod].positions[j];

      t_board.rods[from_rod].occupancy_val[i] = -1;
      t_board.rods[to_rod].occupancy_val[j] = active_disc.disc_index;
}

Vector3 get_inerpolated_coordinate(Vector3 sp, Vector3 tp, double u)
{
      // 4 Control points
      Vector3 p;
      double x_center = (t_board.x_max - t_board.x_min) / 2.0;
      double y_center = (t_board.y_max - t_board.y_min) / 2.0;

      double u3 = u * u * u;
      double u2 = u * u;

      Vector3 cps[4]; //P1, P2, dP1, dP2
```

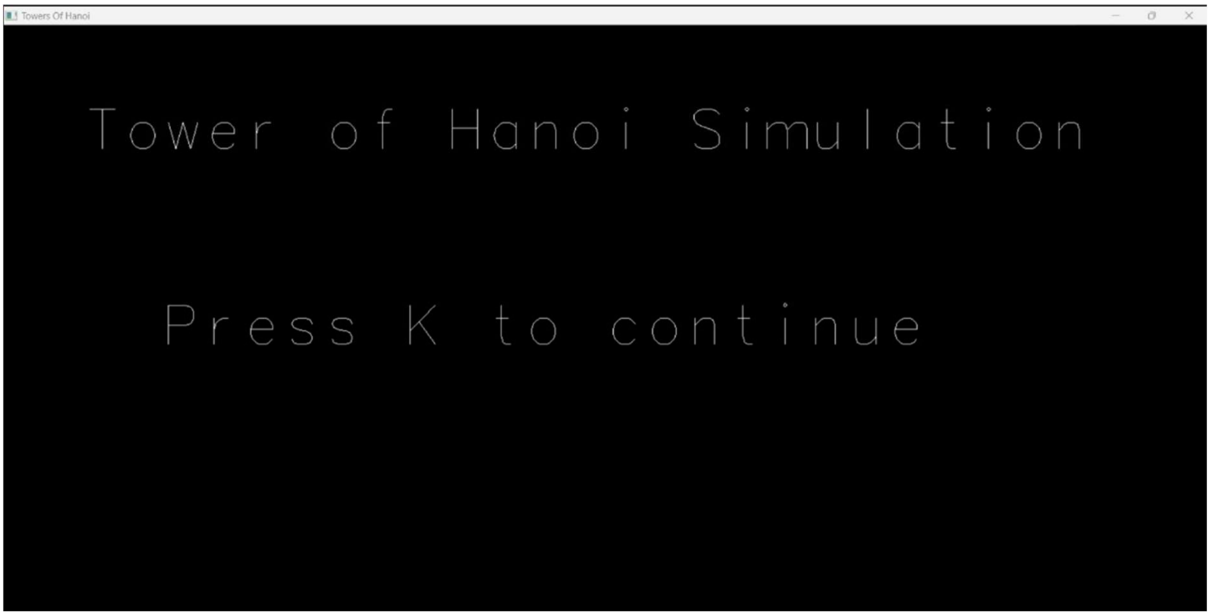
```
// Hermite Interpolation
// Equation of spline
{
	//P1
	cps[0].x = sp.x;
	cps[0].y = y_center;
	cps[0].z = ROD_HEIGHT + 0.2 * (t_board.rod_base_rad);

	//P2
	cps[1].x = tp.x;
	cps[1].y = y_center;
	cps[1].z = ROD_HEIGHT + 0.2 * (t_board.rod_base_rad);

	//dP1
	cps[2].x = (sp.x + tp.x) / 2.0 - sp.x;
	cps[2].y = y_center;
	cps[2].z = 2 * cps[1].z;

	//dP2
	cps[3].x = tp.x - (tp.x + sp.x) / 2.0;
	cps[3].y = y_center;
	cps[3].z = -cps[2].z;

	double h0 = 2 * u3 - 3 * u2 + 1;
	double h1 = -2 * u3 + 3 * u2;
	double h2 = u3 - 2 * u2 + u;
	double h3 = u3 - u2;

	p.x = h0 * cps[0].x + h1 * cps[1].x + h2 * cps[2].x + h3 * cps[3].x;
	p.y = h0 * cps[0].y + h1 * cps[1].y + h2 * cps[2].y + h3 * cps[3].y;
	p.z = h0 * cps[0].z + h1 * cps[1].z + h2 * cps[2].z + h3 * cps[3].z;

}
```

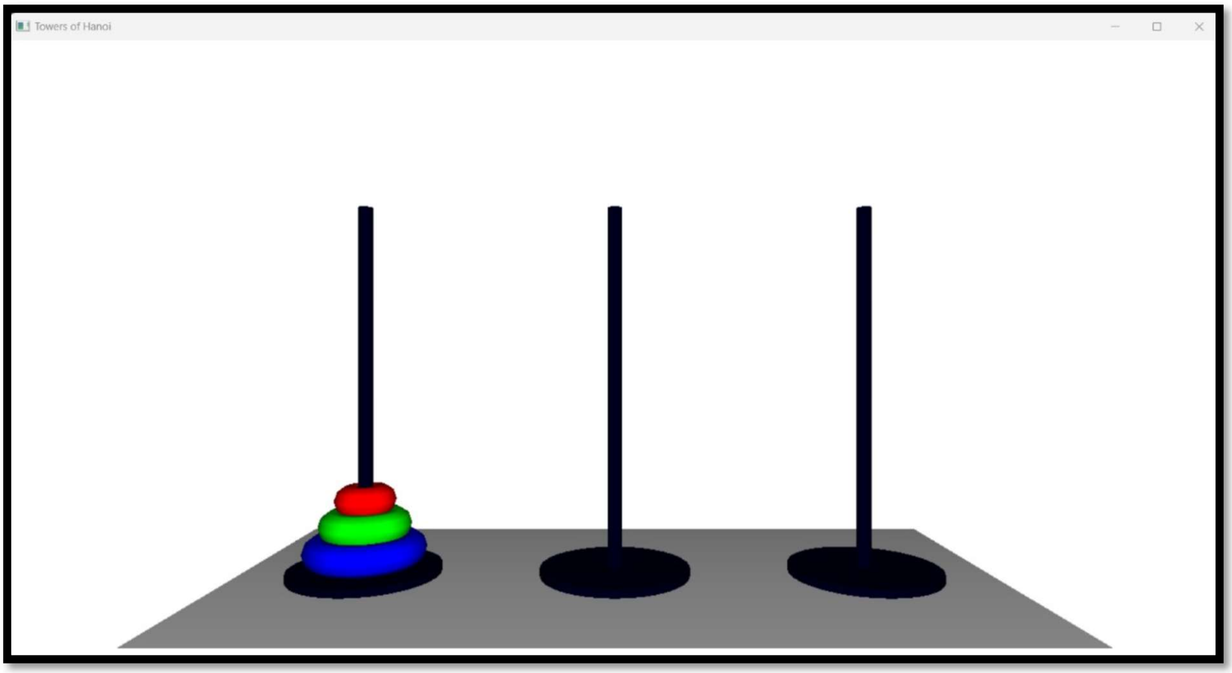
```
    return p;
}

// Normalize function for a vector
void normalize(Vector3& v)
{
    double length = sqrt(v.x * v.x + v.y * v.y + v.z * v.z);
    if (length == 0.0) return;
    v.x /= length;
    v.y /= length;
    v.z /= length;
}

Vector3 operator-(Vector3 const& v1, Vector3 const& v2)
{
    return Vector3(v1.x - v2.x, v1.y - v2.y, v1.z - v2.z);
}

void anim_handler()
{
    FPS = int(30 * SPEED);
    int curr_time = glutGet(GLUT_ELAPSED_TIME);
    int elapsed = curr_time - prev_time;                // in ms
    if (elapsed < 1000 / FPS) return;

    prev_time = curr_time;

    if (to_solve && active_disc.is_in_motion == false) {
        solution_pair s = sol.front();

        cout << s.f << ", " << s.t << endl;

        sol.pop_front();
        int i;
```

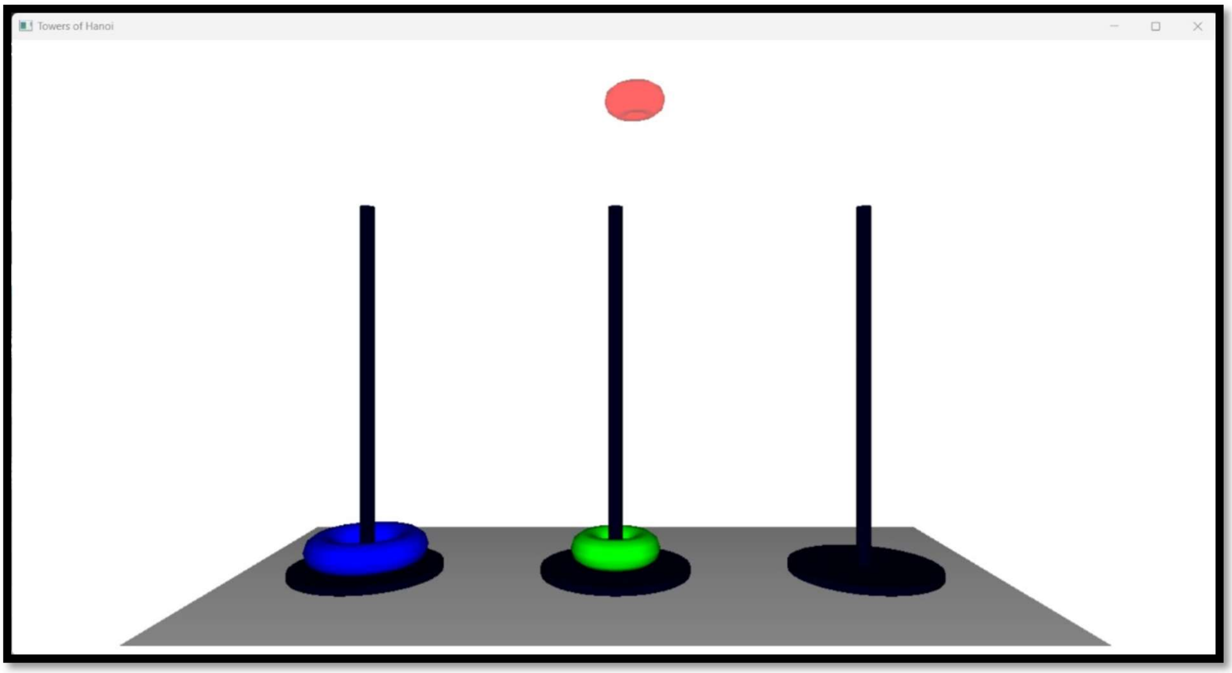
```
		for (i = NUM_DISCS; i >= 0 && t_board.rods[s.f].occupancy_val[i] < 0; i--);
		int ind = t_board.rods[s.f].occupancy_val[i];

		if (ind >= 0)
			active_disc.disc_index = ind;

		move_disc(s.f, s.t);
		if (sol.size() == 0)
			to_solve = false;
	}

	if (active_disc.is_in_motion)
	{
		int ind = active_disc.disc_index;
		ActiveDisc& ad = active_disc;

		if (ad.u == 0.0 && (discs[ind].position.z < ROD_HEIGHT + 0.2 *
(t_board.rod_base_rad)))
		{
			discs[ind].position.z += 0.05;
			glutPostRedisplay();
			return;
		}

		static bool done = false;
		if (ad.u == 1.0 && discs[ind].position.z > ad.dest_pos.z)
		{
			done = true;
			discs[ind].normal = Vector3(0, 0, 1);
			discs[ind].position.z -= 0.05;
			glutPostRedisplay();
			return;
		}
```

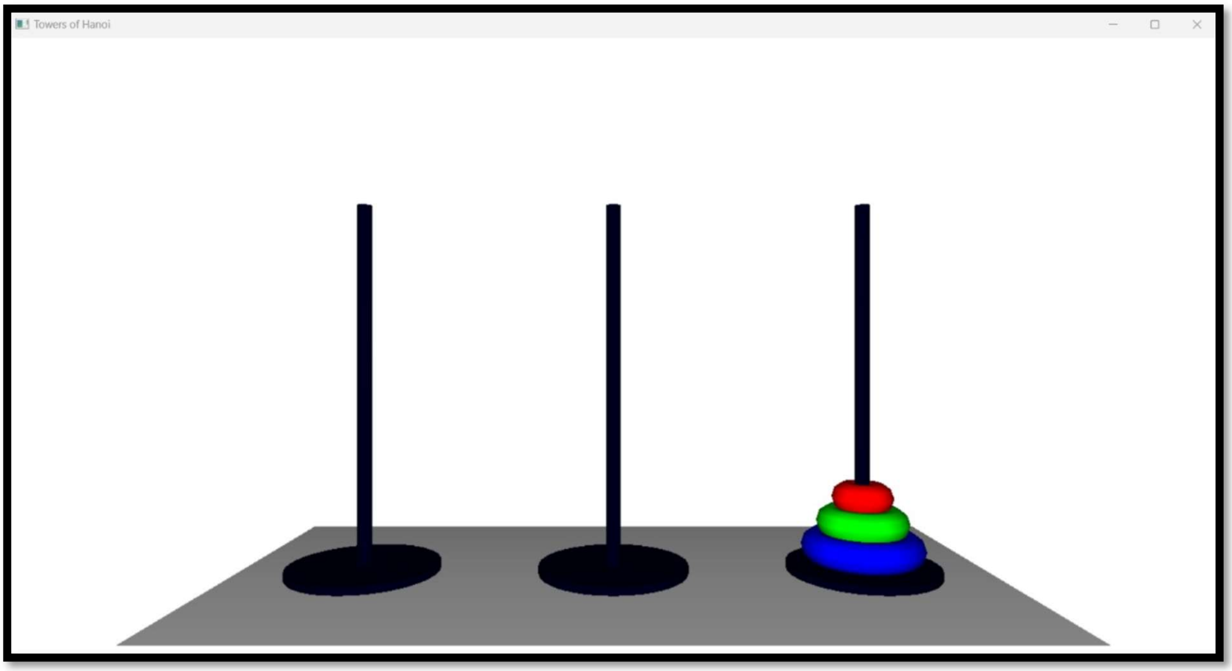
```
			ad.u += ad.step_u;
			if (ad.u > 1.0) {
				ad.u = 1.0;
			}

			if (!done) {
				Vector3 prev_p = discs[ind].position;
				Vector3 p = get_inerpolated_coordinate(ad.start_pos, ad.dest_pos,
ad.u);
				discs[ind].position = p;
				discs[ind].normal.x = (p - prev_p).x;
				discs[ind].normal.y = (p - prev_p).y;
				discs[ind].normal.z = (p - prev_p).z;
				normalize(discs[ind].normal);
			}

			if (ad.u >= 1.0 && discs[ind].position.z <= ad.dest_pos.z) {
				discs[ind].position.z = ad.dest_pos.z;
				ad.is_in_motion = false;
				done = false;
				ad.u = 0.0;
				discs[ad.disc_index].normal = Vector3(0, 0, 1);
				ad.disc_index = -1;
			}
			glutPostRedisplay();
		}
}
```

SCREENSHOTS:






Towers of Hanoi


Towers of Hanoi