# FR CONCEICAO RODRIGUES INSTITUTE OF TECHNOLOGY

# VASHI, NAVI MUMBAI – 400076

**MINI PROJECT**

**SUBJECT: *COMPUTER GRAPHICS***

**TOPIC: *TOWERS OF HANOI***

**DATE: 23 APRIL 2011**

**PROJECT GROUP:**

**KALPESH U ADHATRAO 100901**

**ADITYA A GAYKAR 100919**

**VIPUL K HONRAO 100921**

**ROHIT JHA 100923**

**ABSTRACT**

The Tower of Hanoi or Towers of [Hanoi](http://en.wikipedia.org/wiki/Hanoi) , also called the Tower of Brahma or Towers of Brahma, is a [mathematical game](http://en.wikipedia.org/wiki/Mathematical_game) or [puzzle](http://en.wikipedia.org/wiki/Puzzle). It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following rules:

* Only one disk may be moved at a time.
* Each move consists of taking the upper disk from one of the rods and sliding it onto another rod, on top of the other disks that may already be present on that rod.
* No disk may be placed on top of a smaller disk.

This project implements the Towers of Hanoi in C using OpenGL. It demonstrates features such as rotation and translation in space, lighting, shading, perspective and orthogonal views, fog, back wall, etc.

The program uses a menu for displaying the effects of these features. The menu can be accessed by right-clicking on the screen at runtime. The program by default is paused. It can be started by right-clicking and selecting “Solve”. This causes the program to start executing. The rings of the towers are then moved from one tower to another. At any time of execution, the features can be enabled or disabled.

**DESCRIPTION**

The primary requirements for this program are to declare constants for the viewport such as screen width (*WIDTH*) and height (*HEIGHT*). The number of disks/rings for which the program is to be solved is declared in a global variable *NUM\_DISKS*. By default this is set to 20. Other global constants defined are for a disk’s height (*DISK\_HEIGHT*), and other run-time options such as Solve (*HANOI\_SOLVE*), Exit (*HANOI\_QUIT*), Lighting settings (*HANOI\_LIGHTING*), Back-Wall settings (*HANOI\_WALL*) and Fog (*HANOI\_FOG*). Besides these, several OpenGL integral constants for rotation (*xangle, yangle, xlangle, ylangle*) and Boolean constants (*motion, back\_wall*) are defined. For the purpose of lighting, we require direction vectors and position vectors for every light source. Also we need to set the ambient lighting.

After these constants, structures *stack* and *stack\_node* are defined. The *stack* structure stores the disks in the three towers, whereas the *stack\_node* structure links the topmost disk of every tower to other towers (*stack*s). As we use a stack data structure, we define the *push* and *pop* functions for the movement of disks.

The next function we define is *init*. In this function, we create the stacks and set the tail pointers to null. Then we display the three cones for the towers (from the *DrawPosts* function), before which the rings are made by the solid geometric structure call a Torus (from the function *DrawDisks)*. After this, the draw function is called. As the program depends on the user’s choices at runtime, we also declare some exit conditions. A user can exit the program at any time by pressing any of the following keys: ‘Q’, ‘q’, ‘Esc’. These are passed to the function *keyboard*. The user can choose to implement other features such as switching on/off the lighting, fogging the display, displaying the back-wall and exiting by right-clicking on the screen and selecting the desired option. All this is handled by the function *hanoi\_menu*, which uses a switch statement to perform the selected option.

For solving, we use the recursive procedure for the Towers of Hanoi problem. This is implemented in the recursive function *mov*. After every movement the entire screen is redrawn in the method *update*, which calls the glutPostRedisplay() function. This translates the topmost disk from one tower to another.

The rotation can be performed by the user during run-time by using the mouse. For this, the user is required to left-click on the screen. The program then finds the location where the left mouse button was pressed down and passes the coordinates to the function *hanoi\_mouse*. Then it continuously tracks the motion of the mouse’s movements and using these values, rotates the entire set of towers and disks about the origin, which is the screen’s center.

The *main* function calls the *glutInit* function to create the window and all the elements such as display-mode (RGBA), enabling the lighting settings, creating the menu, etc. Then the initial call to the function *mov* is called, which starts the execution of the program.

**SOURCE CODE**

#include <stdlib.h>

#include <GL/glut.h>

#include <stdio.h>

#include <string.h>

double WIDTH = 1360;

double HEIGHT = 768;

int NUM\_DISKS = 20;

GLboolean motion = GL\_FALSE;

GLboolean back\_wall = GL\_FALSE;

GLint xangle = 0, yangle = 0;

GLint xlangle = 0, ylangle = 0;

#define other(i,j) (6-(i+j))

#define wallz -(WIDTH/2)

#define DISK\_HEIGHT 20

#define CONE NUM\_DISKS+1

#define WALL CONE + 1

#define HANOI\_SOLVE 0

#define HANOI\_QUIT 1

#define HANOI\_LIGHTING 2

#define HANOI\_WALL 3

#define HANOI\_FOG 4

GLfloat lightOneDirection[] = {-0.1, 0, -1};

GLfloat lightOnePosition[] = {450, 100, 1500, 1};

GLfloat lightOneColor[] = {1.0, 0.5, 0, 1.0};

GLfloat lightTwoDirection[] = {0, 0.05, -1};

GLfloat lightTwoPosition[] = {900, 100, 1500, 1};

GLfloat lightTwoColor[] = {1.0, 0.0, 1, 1.0};

GLfloat lightZeroPosition[] = {400, 200, 300, 1};

GLfloat lightZeroColor[] = {.3, .3, .3, .3};

GLfloat diskColor[] = {1.0, 0.5, 0.5, .1}, poleColor[] = {0.5, 0.5, 0.5, 1.0};

//Structures for stack

typedef struct stack\_node

{

int size;

struct stack\_node \*next;

} stack\_node;

typedef struct stack

{

struct stack\_node \*head;

int depth;

} stack;

stack poles[4];

void push(int which, int size)

{

stack\_node \*new = malloc(sizeof(stack\_node));

if (!new)

{

fprintf(stderr, "out of memory!\n");

exit(-1);

}

new->size = size;

new->next = poles[which].head;

poles[which].head = new;

poles[which].depth++;

}

int pop(int which)

{

int retval = poles[which].head->size;

stack\_node \*temp = poles[which].head;

poles[which].head = poles[which].head->next;

poles[which].depth--;

free(temp);

return retval;

}

typedef struct move\_node

{

int t, f;

struct move\_node \*next;

struct move\_node \*prev;

} move\_node;

typedef struct move\_stack

{

int depth;

struct move\_node \*head, \*tail;

} move\_stack;

move\_stack moves;

//Initialize towers and rings

void init(void)

{

int i;

for (i = 0; i < 4; i++)

{

poles[i].head = NULL;

poles[i].depth = 0;

}

moves.head = NULL;

moves.tail = NULL;

moves.depth = 0;

for (i = 1; i <= NUM\_DISKS; i++)

{

glNewList(i, GL\_COMPILE);

{

glutSolidTorus(DISK\_HEIGHT / 2, 5 \* i, 15, 15);

}

glEndList();

}

glNewList(CONE, GL\_COMPILE);

{

glutSolidCone(10, (NUM\_DISKS + 1) \* DISK\_HEIGHT, 20, 20);

}

glEndList();

}

void mpop(void)

{

move\_node \*temp = moves.head;

moves.head = moves.head->next;

free(temp);

moves.depth--;

}

void mpush(int t, int f)

{

move\_node \*new = malloc(sizeof(move\_node));

if (!new)

{

fprintf(stderr, "Out of memory!\n");

exit(-1);

}

new->t = t;

new->f = f;

new->next = NULL;

new->prev = moves.tail;

if (moves.tail)

moves.tail->next = new;

moves.tail = new;

if (!moves.head)

moves.head = moves.tail;

moves.depth++;

}

//Exit conditions

void keyboard(unsigned char key, int x, int y)

{

switch (key)

{

case 27: /\* ESC \*/

case 'q':

case 'Q':

exit(0);

}

}

void update(void)

{

glutPostRedisplay();

}

void DrawPost(int xcenter)

{

glPushMatrix();

{

glTranslatef(xcenter, 0, 0);

glRotatef(90, -1, 0, 0);

glCallList(CONE);

}

glPopMatrix();

}

void DrawPosts(void)

{

glColor3fv(poleColor);

glLineWidth(10);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, poleColor);

DrawPost((int)(WIDTH / 4));

DrawPost((int)(2 \* WIDTH / 4));

DrawPost((int)(3 \* WIDTH / 4));

}

void DrawDisk(int xcenter, int ycenter, int size)

{

glPushMatrix();

{

glTranslatef(xcenter, ycenter, 0);

glRotatef(90, 1, 0, 0);

glCallList(size);

} glPopMatrix();

}

void DrawDooDads(void)

{

int i;

stack\_node \*temp;

int xcenter, ycenter;

glColor3fv(diskColor);

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, diskColor);

for (i = 1; i <= 3; i++)

{

xcenter = i \* WIDTH / 4;

for (temp = poles[i].head, ycenter = DISK\_HEIGHT \* poles[i].depth - DISK\_HEIGHT / 2; temp; temp = temp->next, ycenter -= DISK\_HEIGHT) {

DrawDisk(xcenter, ycenter, temp->size);

}

}

}

#define MOVE(t,f) mpush((t),(f))

static void mov(int n, int f, int t)

{

int o;

if (n == 1)

{

MOVE(t, f);

return;

}

o = other(f, t);

mov(n - 1, f, o);

mov(1, f, t);

mov(n - 1, o, t);

}

GLfloat wallcolor[] = {0, 0.5, 0.5, .1};

void DrawWall(void)

{

int i, j;

glColor3fv(wallcolor);

for (i = 0; i < WIDTH; i += 10)

{

for (j = 0; j < HEIGHT; j += 10)

{

glBegin(GL\_POLYGON);

{

glNormal3f(0, 0, 1);

glVertex3f(i + 10, j, wallz);

glVertex3f(i + 10, j + 10, wallz);

glVertex3f(i, j + 10, wallz);

glVertex3f(i, j, wallz);

}

glEnd();

}

}

}

void draw(void)

{

int t, f;

glClear(GL\_COLOR\_BUFFER\_BIT | GL\_DEPTH\_BUFFER\_BIT);

if (back\_wall)

{

glMaterialfv(GL\_FRONT, GL\_DIFFUSE, wallcolor);

DrawWall();

}

glPushMatrix();

{

glTranslatef(WIDTH / 2, HEIGHT / 2, 0);

glRotatef(xlangle, 0, 1, 0);

glRotatef(ylangle, 1, 0, 0);

glTranslatef(-WIDTH / 2, -HEIGHT / 2, 0);

glLightfv(GL\_LIGHT0, GL\_POSITION, lightZeroPosition);

}

glPopMatrix();

glPushMatrix();

{

glTranslatef(WIDTH / 2, HEIGHT / 2, 0);

glRotatef(xangle, 0, 1, 0);

glRotatef(yangle, 1, 0, 0);

glTranslatef(-WIDTH / 2, -HEIGHT / 2, 0);

DrawPosts();

DrawDooDads();

}

glPopMatrix();

if (motion && moves.depth) {

t = moves.head->t;

f = moves.head->f;

push(t, pop(f));

mpop();

}

glutSwapBuffers();

}

void hanoi\_menu(int value)

{

switch (value)

{

case HANOI\_SOLVE:

motion = !motion;

if(motion)

{

glutIdleFunc(update);

}

else

{

glutIdleFunc(NULL);

}

break;

case HANOI\_LIGHTING:

if (glIsEnabled(GL\_LIGHTING))

glDisable(GL\_LIGHTING);

else

glEnable(GL\_LIGHTING);

break;

case HANOI\_WALL:

back\_wall = !back\_wall;

break;

case HANOI\_FOG:

if (glIsEnabled(GL\_FOG))

glDisable(GL\_FOG);

else {

glEnable(GL\_FOG);

glFogi(GL\_FOG\_MODE, GL\_EXP);

glFogf(GL\_FOG\_DENSITY, .01);

}

break;

case HANOI\_QUIT:

exit(0);

break;

}

glutPostRedisplay();

}

int oldx, oldy;

GLboolean leftb = GL\_FALSE, middleb = GL\_FALSE;

void hanoi\_mouse(int button, int state, int x, int y)

{

if (button == GLUT\_LEFT\_BUTTON)

{

oldx = x;

oldy = y;

if (state == GLUT\_DOWN)

leftb = GL\_TRUE;

else

leftb = GL\_FALSE;

}

if (button == GLUT\_MIDDLE\_BUTTON)

{

oldx = x;

oldy = y;

if (state == GLUT\_DOWN)

middleb = GL\_TRUE;

else

middleb = GL\_FALSE;

}

}

void hanoi\_visibility(int state)

{

if (state == GLUT\_VISIBLE && motion)

{

glutIdleFunc(update);

}else

{

glutIdleFunc(NULL);

}

}

void hanoi\_motion(int x, int y)

{

if (leftb) {

xangle -= (x - oldx);

yangle -= (y - oldy);

}

if (middleb) {

xlangle -= (x - oldx);

ylangle -= (y - oldy);

}

oldx = x;

oldy = y;

glutPostRedisplay();

}

int main(int argc, char \*argv[])

{

int i;

glutInit(&argc, argv);

for(i=1; i<argc; i++)

{

if(!strcmp("-n", argv[i]))

{

i++;

if(i >= argc)

{

printf("hanoi: number after -n is required\n");

exit(1);

}

NUM\_DISKS = atoi(argv[i]);

}

}

glutInitWindowSize((int)WIDTH, (int)HEIGHT);

glutInitDisplayMode(GLUT\_RGBA | GLUT\_DOUBLE | GLUT\_DEPTH);

glutCreateWindow("Hanoi");

glutDisplayFunc(draw);

glutKeyboardFunc(keyboard);

glViewport(0, 0, (int)WIDTH, (int)HEIGHT);

glMatrixMode(GL\_PROJECTION);

glLoadIdentity();

glOrtho(0, WIDTH, 0, HEIGHT, -10000, 10000);

glMatrixMode(GL\_MODELVIEW);

glLoadIdentity();

glClearColor(0, 0, 0, 0);

glClearDepth(1.0);

glEnable(GL\_CULL\_FACE);

glEnable(GL\_DEPTH\_TEST);

glLightModeli(GL\_LIGHT\_MODEL\_LOCAL\_VIEWER, 1);

glLightfv(GL\_LIGHT1, GL\_POSITION, lightOnePosition);

glLightfv(GL\_LIGHT1, GL\_DIFFUSE, lightOneColor);

glLightf(GL\_LIGHT1, GL\_SPOT\_CUTOFF, 10);

glLightfv(GL\_LIGHT1, GL\_SPOT\_DIRECTION, lightOneDirection);

glEnable(GL\_LIGHT1);

glLightfv(GL\_LIGHT2, GL\_POSITION, lightTwoPosition);

glLightfv(GL\_LIGHT2, GL\_DIFFUSE, lightTwoColor);

//glLightf(GL\_LIGHT2,GL\_LINEAR\_ATTENUATION,.005);

glLightf(GL\_LIGHT2, GL\_SPOT\_CUTOFF, 10);

glLightfv(GL\_LIGHT2, GL\_SPOT\_DIRECTION, lightTwoDirection);

glEnable(GL\_LIGHT2);

glLightfv(GL\_LIGHT0, GL\_DIFFUSE, lightZeroColor);

glEnable(GL\_LIGHT0);

glEnable(GL\_LIGHTING);

glutMouseFunc(hanoi\_mouse);

glutMotionFunc(hanoi\_motion);

glutVisibilityFunc(hanoi\_visibility);

glutCreateMenu(hanoi\_menu);

glutAddMenuEntry("Solve", HANOI\_SOLVE);

glutAddMenuEntry("Lighting", HANOI\_LIGHTING);

glutAddMenuEntry("Back Wall", HANOI\_WALL);

glutAddMenuEntry("Fog", HANOI\_FOG);

glutAddMenuEntry("Quit", HANOI\_QUIT);

glutAttachMenu(GLUT\_RIGHT\_BUTTON);

init();

for (i = 0; i < NUM\_DISKS; i++)

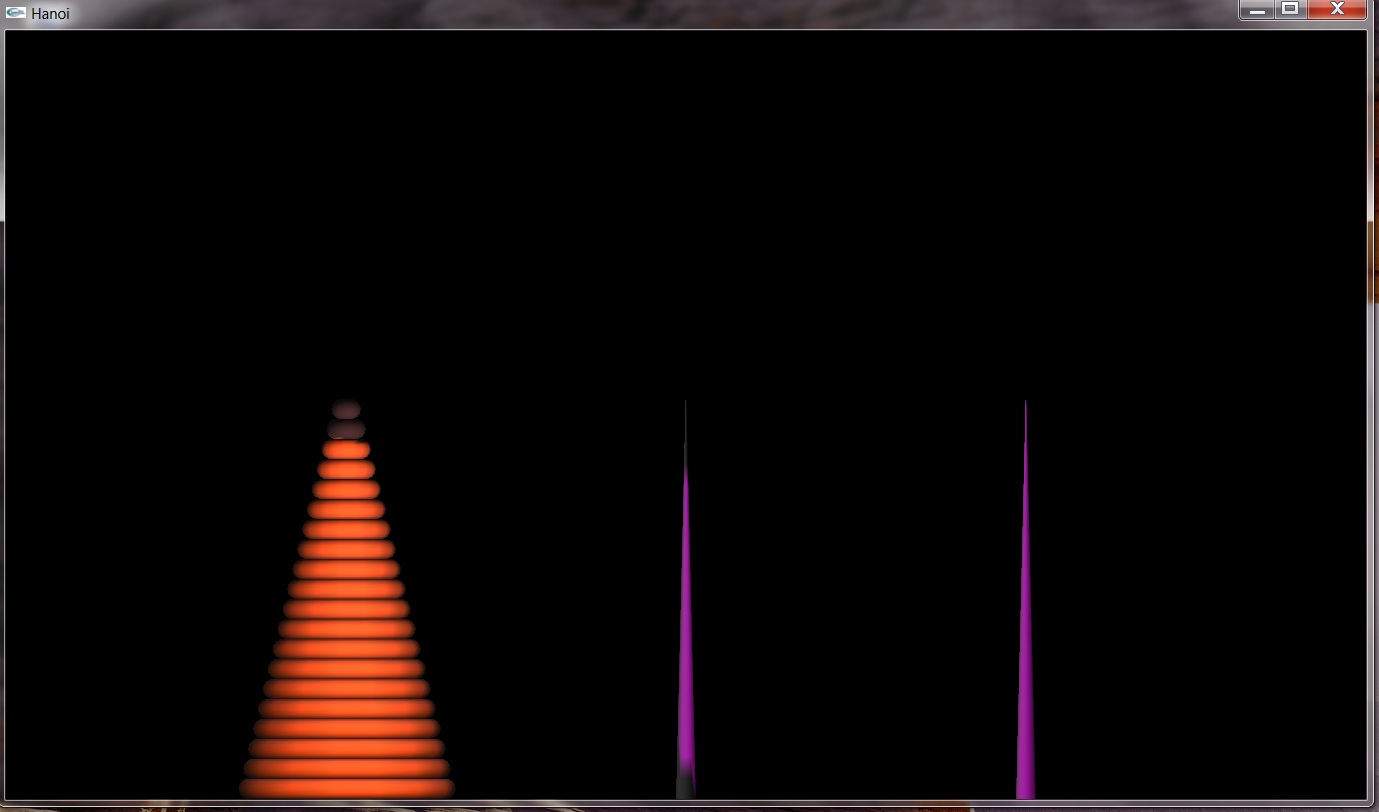
push(1, NUM\_DISKS - i);

mov(NUM\_DISKS, 1, 3);

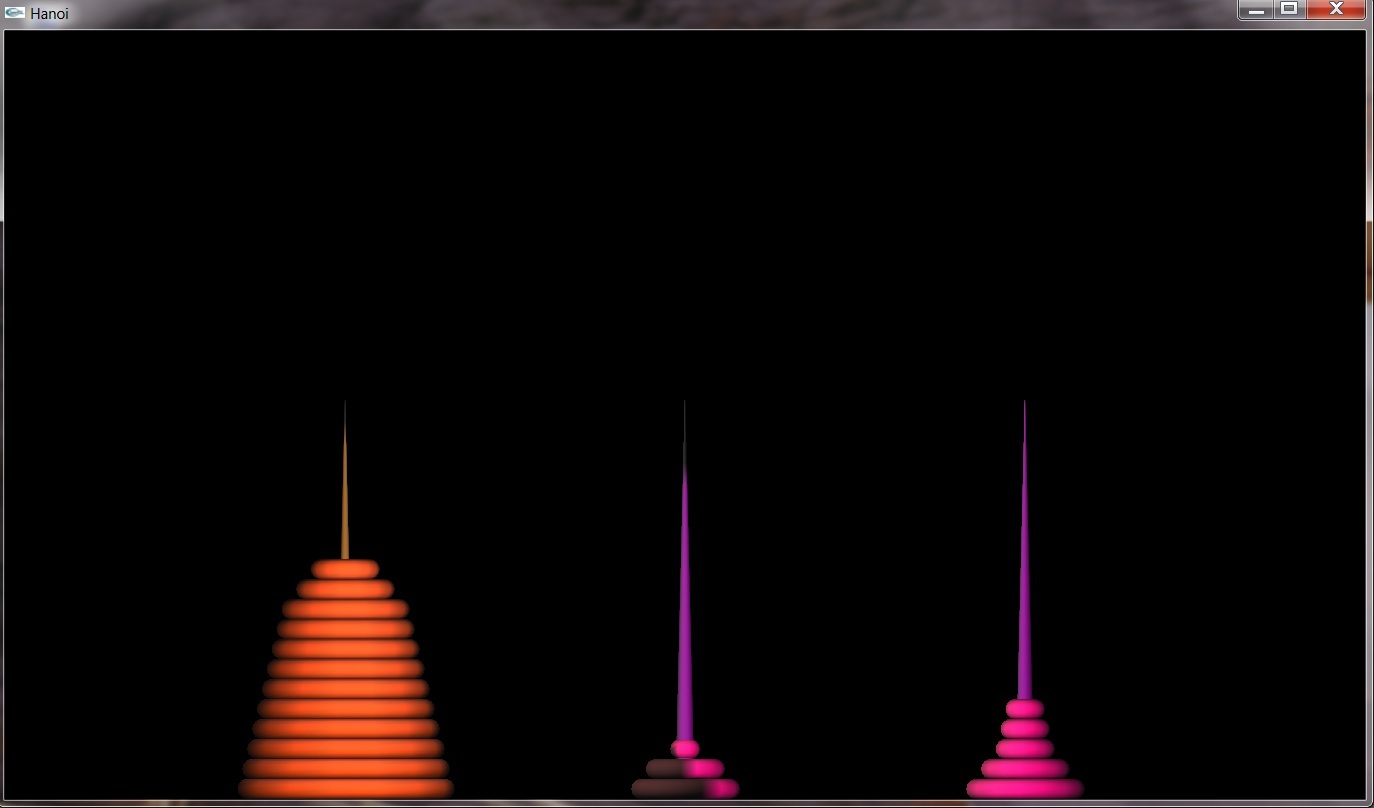
glutMainLoop();

return 0;

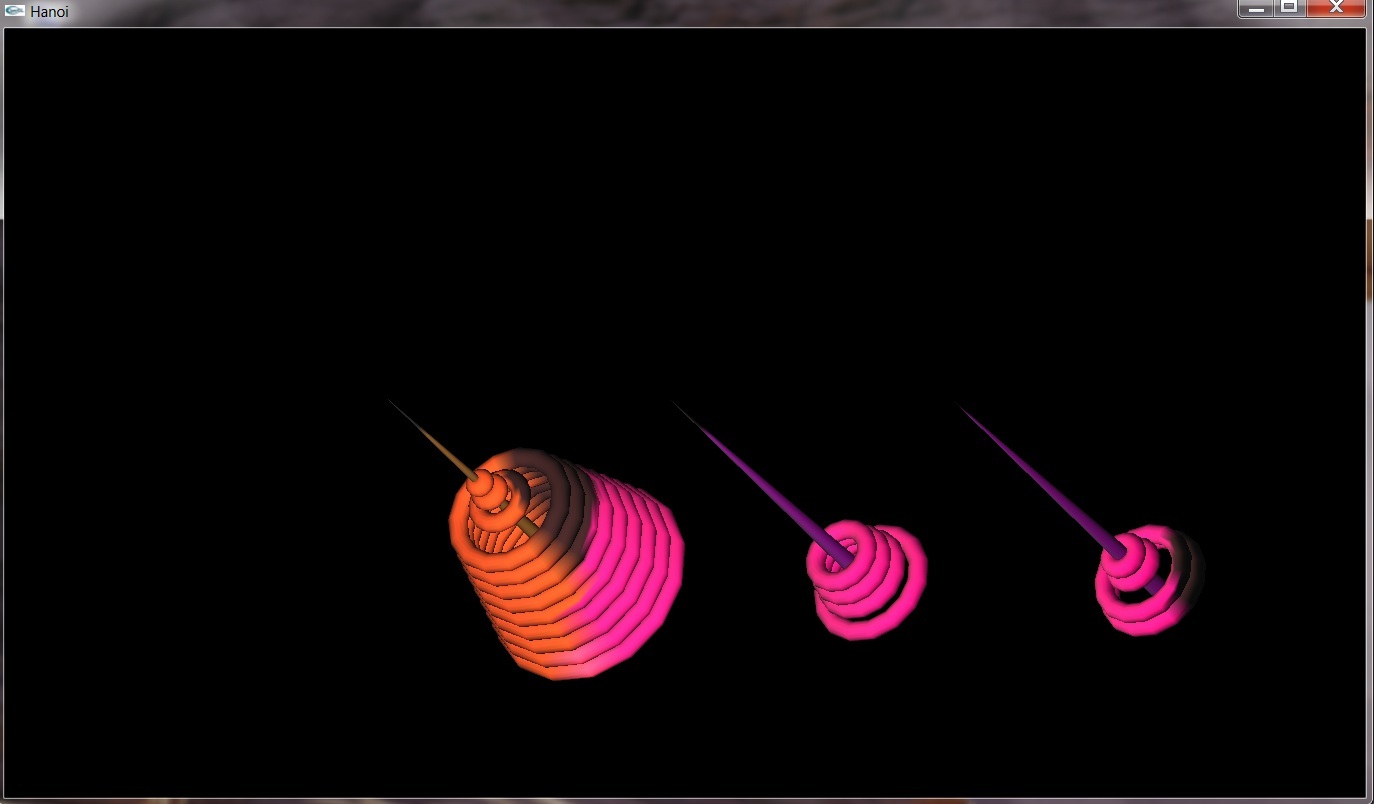
}



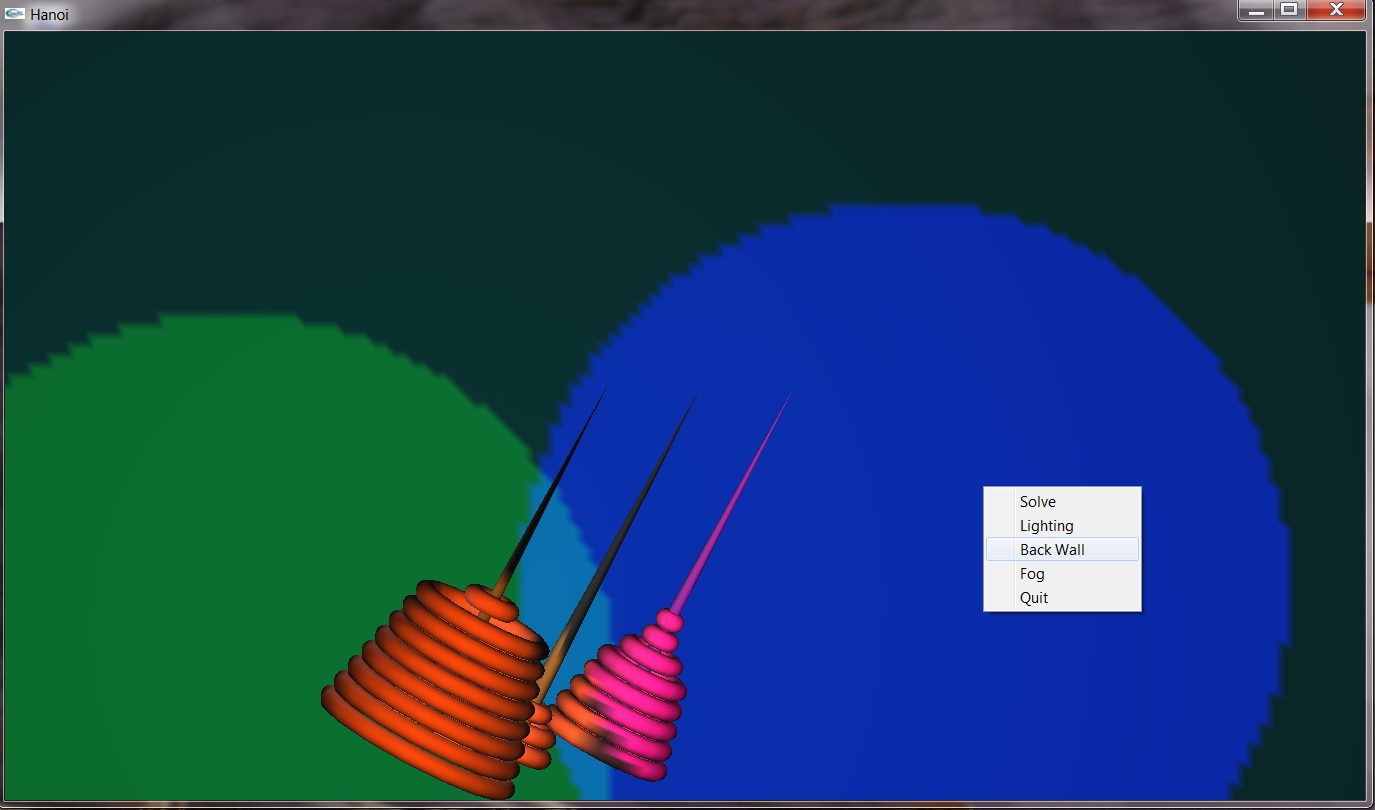
1. Starting state

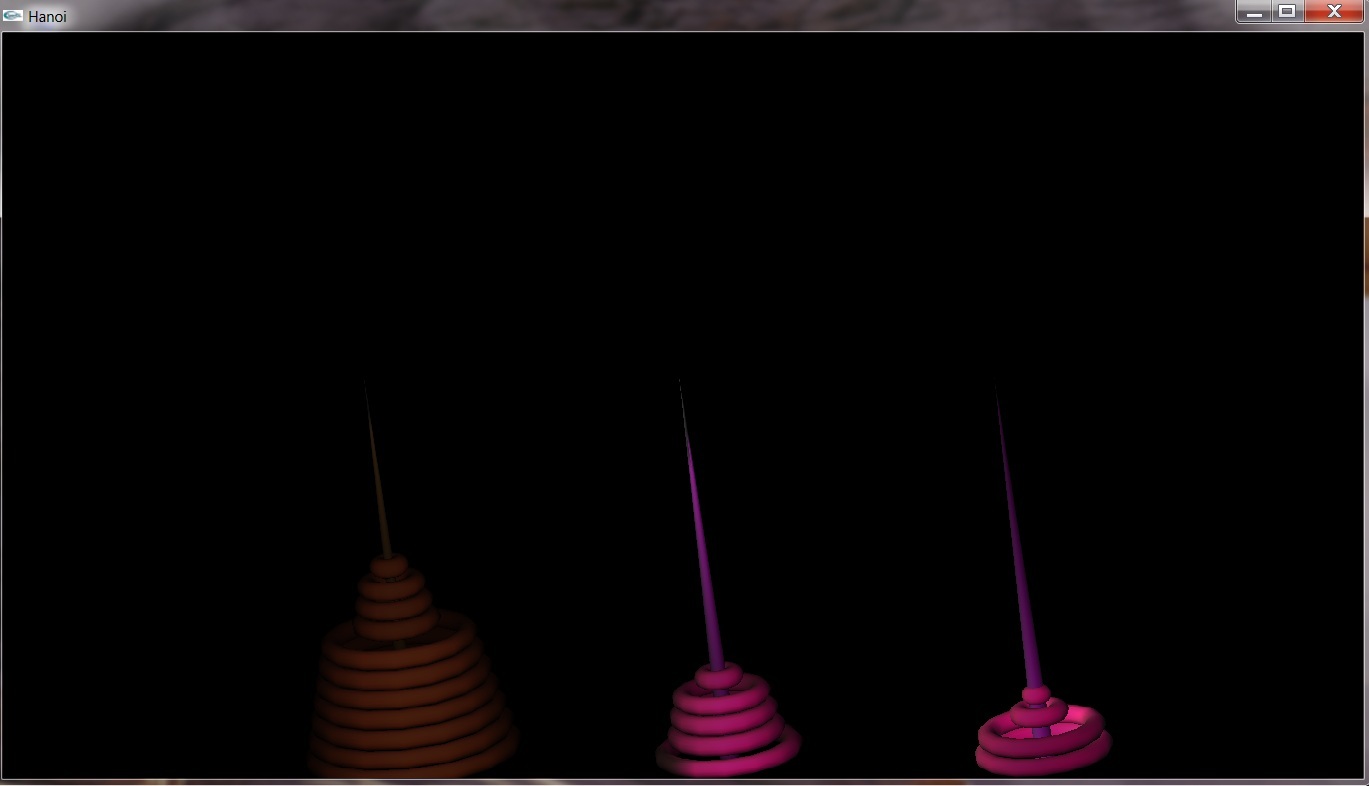


1. Intermediate state

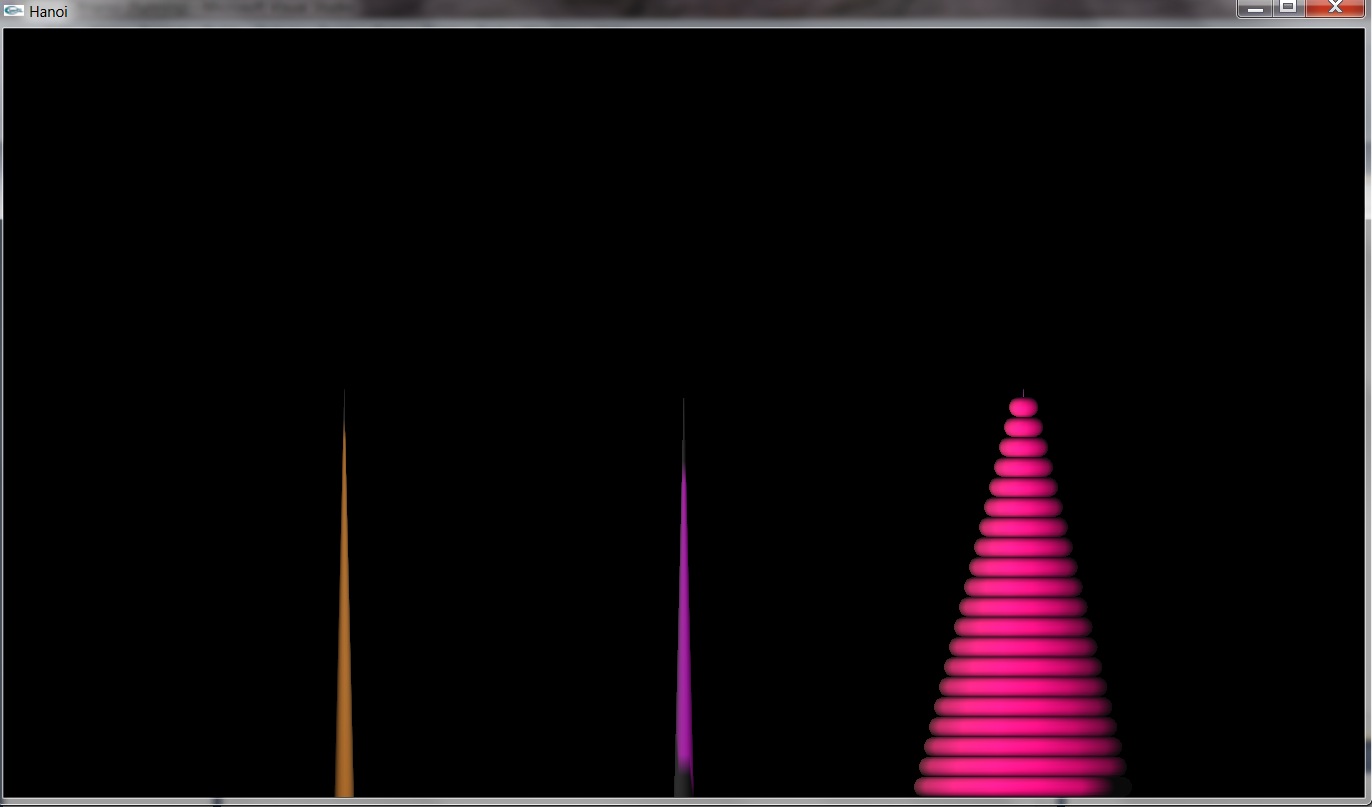


1. (Above) Rotated View
2. (Below) A Rotated View with the Back Wall displayed





1. View with Fog On



1. Finished state