CHAPTER 1

INTRODUCTION

Open GL:

OpenGL is a software interface to graphics hardware. This interface consists of about 150 distinct commands that you use to specify the objects and operations needed to produce interactive threedimensional applications.

OpenGL is designed as a streamlined, hardware-independent interface to be implemented on many different hardware platforms. To achieve these qualities, no commands for performing windowing tasks or obtaining user input are included in OpenGL; instead, you must work through whatever windowing system controls the particular hardware you're using. Similarly, OpenGL doesn't provide high-level commands for describing models of three-dimensional objects. Such commands might allow you to specify relatively complicated shapes such as automobiles, parts of the body, airplanes, or molecules. With OpenGL, you must build up your desired model from a small set of geometric primitives - points, lines, and polygons.

A sophisticated library that provides these features could certainly be built on top of OpenGL. The OpenGL Utility Library (GLU) provides many of the modeling features, such as quadric surfaces and NURBS (Non-Uniform Rational B-Splines) curves and surfaces. GLU is a standard part of every OpenGL implementation. Also, there is a higher-level, object-oriented toolkit, Open Inventor, which is built atop OpenGL, and is available separately for many implementations of OpenGL.

OpenGL-Related Libraries:

OpenGL provides a powerful but primitive set of rendering commands, and all higher-level drawing must be done in terms of these commands. Also, OpenGL programs have to use the underlying mechanisms of the windowing system. A number of libraries exist to allow you to simplify your programming tasks, including the following: • The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations

and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of every OpenGL implementation. GLU routines use the prefix glu. • For every window system, there is a library that extends the functionality of that window system to support OpenGL rendering. For machines that use the X Window System, the OpenGL Extension to the X Window System (GLX) is provided as an adjunct to

OpenGL. GLX routines use the prefix glX. For Microsoft Windows, the WGL routines provide the Windows to OpenGL interface. All WGL routines use the prefix wgl. For IBM OS/2, the PGL is the Presentation Manager to OpenGL interface, and its routines use the prefix pgl. • The OpenGL Utility Toolkit (GLUT) is a window system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window system APIs. GLUT routines use the prefix glut. • Open Inventor is an object-oriented toolkit based on OpenGL which provides objects and methods for creating interactive three-dimensional graphics applications. Open Inventor, which is written in C++, provides prebuilt objects and a built-in event model for user interaction, high-level application components for creating and editing three-dimensional scenes, and the ability to print objects and exchange data in other graphics formats. Open Inventor is separate from OpenGL.

Include Files:

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which requires inclusion of the glu.h header file. So almost every OpenGL source file begins with #include #include If you are directly accessing a window interface library to support OpenGL, such as GLX, AGL, PGL, or WGL, you must include additional header files. For example, if you are calling GLX, you may need to add these lines to your code

#include<GL/glut.h>

Note that glut.h includes gl.h, glu.h, and glx.h automatically, so including all three files is redundant. GLUT for Microsoft Windows includes the appropriate header file to access WGL.

WSL [Window Subsystem for Linux]:

Windows Subsystem for Linux (WSL) is a feature of Windows that allows developers to run a Linux environment without the need for a separate virtual machine or dual booting. There are two versions of WSL: WSL 1 and WSL 2. WSL 1 was first released on August 2, 2016, and acts as a compatibility layer for running Linux binary executables (in ELF format) by implementing Linux system calls on the Windows kernel.[3] It is available on Windows 10, Windows 10 LTSB/LTSC, Windows 11,[4] Windows Server 2016, Windows Server 2019 and Windows Server 2022.

In May 2019, WSL 2 was announced,[5] introducing important changes such as a real Linux kernel,[6] through a subset of Hyper-V features. WSL 2 differs from WSL 1 in that WSL 2 runs inside a managed virtual machine that implements the full Linux kernel. As a result, WSL 2 is compatible with more Linux binaries than WSL 1, as not all sisals were implemented in WSL 1**.** June

2019, Since WSL 2 is available to Windows 10 customers through the Windows Insider program, including the Home edition.[7] WSL is not available to all Windows 10 users by default. It can be installed either by joining the Windows Insider program or manual install

History:

Microsoft's first foray into achieving Unix-like compatibility on Windows began with the Microsoft POSIX Subsystem, superseded by Windows Services for UNIX via MKS/Interix, which was eventually deprecated with the release of Windows 8.1. The technology behind Windows Subsystem for Linux originated in the unreleased Project Astoria, which enabled some Android applications to run on Windows 10 Mobile.[9] It was first made available in Windows 10 Insider Preview build 14316.[10]

Whereas Microsoft's previous projects and the third-party Cygwin had focused on creating their own unique Unix-like environments based on the POSIX standard, WSL aims for native Linux compatibility. Instead of wrapping non-native functionality into Win32 system calls as Cygwin did, WSL's initial design (WSL 1) leveraged the NT kernel executive to serve Linux programs as special, isolated minimal processes (known as "pico processes") attached to kernel mode "pico providers" as dedicated system call and exception handlers distinct from that of a vanilla NT process, opting to reutilize existing NT implementations wherever possible

WSL beta was introduced in Windows 10 version 1607 (Anniversary Update) on August 2, 2016. Only Ubuntu (with Bash as the default shell) was supported. WSL beta was also called "Bash on Ubuntu on Windows" or "Bash on Windows". WSL was no longer beta in Windows 10 version 1709 (Fall Creators Update), released on October 17, 2017. Multiple Linux distributions could be installed and were available for install in the Windows Store.

n 2017 Richard Stallman expressed fears that integrating Linux functionality into Windows will only hinder the development of free software, calling efforts like WSL "a step backward in the campaign for freedom."[13]

Though WSL (via this initial design) was much faster and arguably much more popular than the previous UNIX-on-Windows projects, Windows kernel engineers found difficulty in trying to increase WSL's performance and syscall compatibility by trying to reshape the existing NT kernel to recognize and operate correctly on Linux's API. At a Microsoft Ignite conference in 2018, Microsoft engineers gave a high-level overview of a new "lightweight" Hyper-V VM technology for containerization where a virtualized kernel could make direct use of NT primitives on the host.[14] In

2019, Microsoft announced a completely redesigned WSL architecture (WSL 2) using this lightweight VM technology hosting actual (customized) Linux kernel images, claiming full syscall compatibility.[6] Microsoft announced WSL 2 on May 6, 2019,[5] and it was shipped with Windows 10 version 2004.[15] It was also backported to Windows 10 version 1903 and 1909.[16]

GPU support for WSL 2 to do GPU-accelerated machine learning was introduced in Windows build 20150.[17] GUI support for WSL 2 to run Linux applications with graphical user interfaces (GUIs) was introduced in Windows build 21364.[18] Both of them are shipped in Windows 11.

In April 2021, Microsoft released a Windows 10 test build that also includes the ability to run Linux graphical user interface (GUI) apps using WSL 2 and CBL-Mariner.[19][18] The Windows Subsystem for Linux GUI (WSLg) was officially released at the Microsoft Build 2021 conference. It is included in Windows 10 Insider build 21364 or later.[20]

Microsoft introduced a Windows Store version of WSL on October 11, 2021, for Windows 11.[21] It reached version 1.0.0 on November 16, 2022 with added support for Windows 10.

Features:

WSL is available in Windows Server 2019 and in versions of Windows 10 from version 1607, though only in 64-bit versions.

Microsoft envisages WSL as "primarily a tool for developers – especially web developers and those who work on or with open source projects".[22] In September 2018, Microsoft said that "WSL requires fewer resources (CPU, memory, and storage) than a full virtual machine" (which prior to WSL was the most direct way to run Linux software in a Windows environment), while also allowing users to use Windows apps and Linux tools on the same set of files.[22]

The first release of WSL provides a Linux-compatible kernel interface developed by Microsoft, containing no Linux kernel code, which can then run the user space of a Linux distribution on top of it, such as Ubuntu,[23][24][25][26] openSUSE,[27] SUSE Linux Enterprise Server,[28][29][12] Debian[30] and Kali Linux.[31] Such a user space might contain a GNU Bash shell and command language, with native GNU command-line tools (sed, awk, etc.), programming-language interpreters (Ruby, Python, etc.), and even graphical applications (using an X11 server at the host side).[22]

The architecture was redesigned in WSL 2,[5] with a Linux kernel running in a lightweight virtual machine environment.

Tower of Hanoi:

The Tower of Hanoi is a mathematical game or puzzle consisting of three rods and a number of disks of various diameters, which can slide onto any rod. The puzzle begins with the disks stacked on one rod in order of decreasing size, the smallest at the top, thus approximating a conical shape. The objective of the puzzle is to move the entire stack to the last 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 stacks and placing it on top of another stack or on an empty rod.

No disk may be placed on top of a disk that is smaller than it.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is 2n − 1, where n is the number of disks

1.Only one disk may be moved at a time.

2.Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack or on an empty rod.

3.No disk may be placed on top of a disk that is smaller than it.

With 3 disks, the puzzle can be solved in 7 moves. The minimal number of moves required to solve a Tower of Hanoi puzzle is 2n − 1, where n is the number of disks.

Origin:

The puzzle was invented by the French mathematician Édouard Lucas in 1883. Numerous myths regarding the ancient and mystical nature of the puzzle popped up almost immediately,[4] including a myth about an Indian temple in Kashi Vishwanath containing a large room with three time-worn posts in it, surrounded by 64 golden disks. But, this story of Indian Kashi Vishwanath temple was spread tongue-in-cheek by a friend of Édouard Lucas.[5]

If the legend were true, and if the priests were able to move disks at a rate of one per second, using the smallest number of moves, it would take them 264 − 1 seconds or roughly 585 billion years to finish,[6] which is about 42 times the estimated current age of the universe.

There are many variations on this legend. For instance, in some tellings, the temple is a monastery, and the priests are monks. The temple or monastery may be in various locales including Hanoi, and may be associated with any religion. In some versions, other elements are introduced, such as the fact that the tower was created at the beginning of the world, or that the priests or monks may make only one move per day.

CHAPTER 2

DESGIN

Recursive Solution:

A key to solving this puzzle is to recognize that it can be solved by breaking the problem down into a collection of smaller problems and further breaking those problems down into even smaller problems until a solution is reached. For example: • label the pegs A, B, C • let n be the total number of discs • number the discs from 1 (smallest, topmost) to n (largest, bottommost) To move n discs from peg A to peg C:

1. move n−1 discs from A to B. This leaves disc n alone on peg A
2. move disc n from A to C
3. move n−1 discs from B to C so they sit on disc n

The above is a recursive algorithm, to carry out steps 1 and 3, apply the same algorithm again for n−1. The entire procedure is a finite number of steps, since at some point the algorithm will be required for n = 1. This step, moving a single disc from peg A to peg C, is trivial. This approach can be given a rigorous mathematical formalism with the theory of dynamic programming and is often used as an example of recursion when teaching programming

Iterative solution:

Animation of an iterative algorithm solving 6-disk problem

A simple solution for the toy puzzle is to alternate moves between the smallest piece and a non-smallest piece. When moving the smallest piece, always move it to the next position in the same direction (to the right if the starting number of pieces is even, to the left if the starting number of pieces is odd). If there is no tower position in the chosen direction, move the piece to the opposite end, but then continue to move in the correct direction. For example, if you started with three pieces, you would move the smallest piece to the opposite end, then continue in the left direction after that. When the turn is to move the non-smallest piece, there is only one legal move. Doing this will complete the puzzle in the fewest moves.

Simpler statement of iterative solution

For an even number of disks:

Make the legal move between pegs A and B (in either direction),

Make the legal move between pegs A and C (in either direction),

Make the legal move between pegs B and C (in either direction),

Repeat until complete.

For an odd number of disks:

Make the legal move between pegs A and C (in either direction),

Make the legal move between pegs A and B (in either direction),

Make the legal move between pegs B and C (in either direction),

Repeat until complete.

After each move check if the C peg is complete. In each case, a total of $2n - 1$ moves are made.

Equivalent iterative solution

Another way to generate the unique optimal iterative solution:

Number the disks 1 through n (largest to smallest).

If n is odd, the first move is from peg A to peg C.

If n is even, the first move is from peg A to peg B.

Now, add these constraints:

No odd disk may be placed directly on an odd disk.

No even disk may be placed directly on an even disk.

There will sometimes be two possible pegs: one will have disks, and the other will be empty. Place the disk on the non-empty peg.

Never move a disk twice in succession.

Considering those constraints after the first move, there is only one legal move at every subsequent turn. The sequence of these unique moves is an optimal solution to the problem equivalent to the iterative solution described above.

Logical analysis

Recursive solution:

As in many mathematical puzzles, finding a solution is made easier by solving a slightly more general problem: how to move a tower of h (height) disks from a starting peg f = A (from) onto a destination peg t = C (to), B being the remaining third peg and assuming t ≠ f. First, observe that the problem is symmetric for permutations of the names of the pegs (symmetric group S3). If a solution is known moving from peg A to peg C, then, by renaming the pegs, the same solution can be used for every other choice of starting and destination peg. If there is only one disk (or even none at all), the problem is trivial. If h = 1, then simply move the disk from peg A to peg C. If h > 1, then somewhere along the sequence of moves, the largest disk must be moved from peg A to another peg, preferably to peg C.

The only situation that allows this move is when all smaller h − 1 disks are on peg B. Hence, first all h − 1 smaller disks must go from A to B. Then move the largest disk and finally move the h − 1 smaller disks from peg B to peg C. The presence of the largest disk does not impede any move of the h − 1 smaller disks and can be temporarily ignored. Now the problem is reduced to moving h − 1 disks from one peg to another one, first from A to B and subsequently from B to C, but the same method can be used both times by renaming the pegs. The same strategy can be used to reduce the h − 1 problem to h − 2, h − 3, and so on until only one disk is left. This is called recursion. This algorithm can be schematized as follows.

Identify the disks in order of increasing size by the natural numbers from 0 up to but not including *h*. Hence disk 0 is the smallest one, and disk *h* − 1 the largest one.

The following is a procedure for moving a tower of *h* disks from a peg **A** onto a peg **C**, with **B** being the remaining third peg:

- If *h* > 1, then first use this procedure to move the *h* − 1 smaller disks from peg **A** to peg **B**.
- Now the largest disk, i.e. disk *h* can be moved from peg **A** to peg **C**.
- If *h* > 1, then again use this procedure to move the *h* − 1 smaller disks from peg **B** to peg **C**.

By means of mathematical induction, it is easily proven that the above procedure requires the minimal number of moves possible and that the produced solution is the only one with this minimal number of moves. Using recurrence relations, the exact number of moves that this solution requires can be calculated by: 2^h-1. This result is obtained by noting that steps 1 and 3 Th-1 take moves, and step 2 takes one move, giving . Th=Th-1.

Non-recursive solution:

The list of moves for a tower being carried from one peg onto another one, as produced by the recursive algorithm, has many regularities. When counting the moves starting from 1, the ordinal of the disk to be moved during move *m* is the number of times *m* can be divided by 2. Hence every odd move involves the smallest disk. It can also be observed that the smallest disk traverses the pegs *f*, *t*, *r*, *f*, *t*, *r*, etc. for odd height of the tower and traverses the pegs *f*, *r*, *t*, *f*, *r*, *t*, etc. for even height of the tower. This provides the following algorithm, which is easier, carried out by hand, than the recursive algorithm.

In alternate moves:

- Move the smallest disk to the peg it has not recently come from.
- Move another disk legally (there will be only one possibility).

For the very first move, the smallest disk goes to peg *t* if *h* is odd and to peg *r* if *h* is even.

Also observe that:

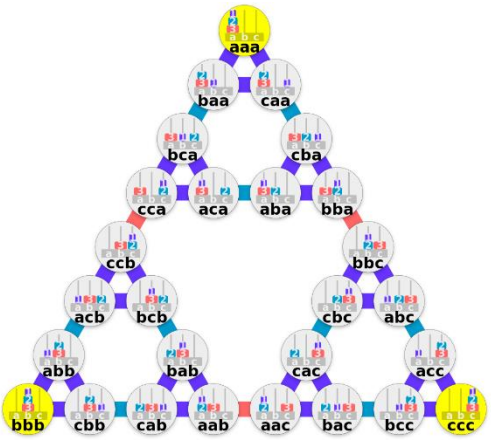
- Disks whose ordinals have even parity move in the same sense as the smallest disk.
- Disks whose ordinals have odd parity move in opposite sense.
- If *h* is even, the remaining third peg during successive moves is *t*, *r*, *f*, *t*, *r*, *f*, etc.
- If *h* is odd, the remaining third peg during successive moves is *r*, *t*, *f*, *r*, *t*, *f*, etc.

With this knowledge, a set of disks in the middle of an optimal solution can be recovered with no more state information than the positions of each disk:

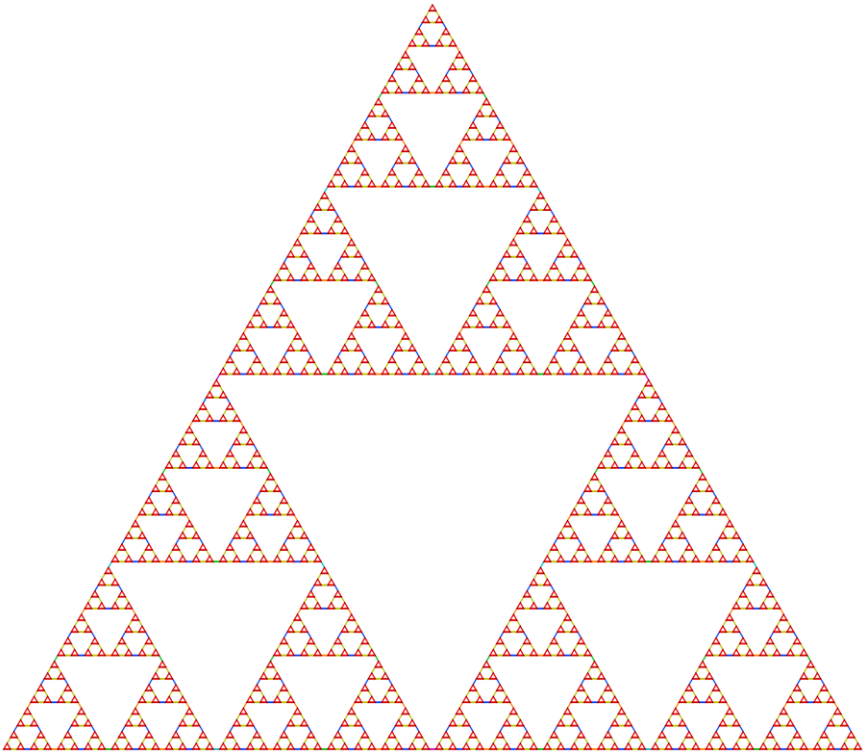
- Call the moves detailed above a disk's "natural" move.
- Examine the smallest top disk that is not disk 0, and note what its only (legal) move would be: if there is no such disk, then we are either at the first or last move.
- If that move is the disk's "natural" move, then the disk has not been moved since the last disk 0 move, and that move should be taken.
- If that move is not the disk's "natural" move, then move disk 0.

Graphical Representation:

The nodes at the vertices of the outermost triangle represent distributions with all disks on the same peg. For h + 1 disks, take the graph of h disks and replace each small triangle with the graph for two disks. For three disks the graph is:



The game graph of level 7 shows the relatedness to the Sierpiński triangle.



Chapter-3

Implementation

Source Code:

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>
#define BREITE 0.1f
#define STANGENBREITE 0.025f
#define SLICES 32
#define INNERSLICES 16
#define LOOPS 1
#define FPS 64   /* more looks nicer, uses more cpu power */
#define FEM 1000.0/FPS
#define FSEM 0.001f   /* speed (bigger is faster)*/

struct config {
                GLfloat gap;
                GLfloat pinradius;
                GLfloat pinheight;
};

struct action {
                char fromstack;
                char tostack;
                struct action* next;
};
typedef struct action action;

struct actions {
                action* head;
                action* tail;
};
typedef struct actions actions;
struct disk {
                char color;
                GLfloat radius;
                struct disk* next;
                struct disk* prev;
};
typedef struct disk disk;

struct stack {
                disk* bottom;
                disk* top;
};
typedef struct stack stack;
```

```
int disks = 3;  //initial number of disks
GLfloat rotX, rotY, zoom, offsetY = 1.5, speed;
GLUquadricObj* quadric;
GLfloat pos;
GLboolean fullscreen;
stack pin[3];
float pinheight[3];
struct config config;
actions actqueue;
action* curaction;
disk* curdisk;
int duration;
char seconds[24] = "Time: 0s";
int draw, maxdraws;

//function prototypes
void moveDisk(int param);
void hanoiinit(void);
void reset();
void Display(void);
void hanoi(actions* queue, const int n, const char pin1, const char pin2, const char pin3);
void push(stack* pin, disk* item);
disk* pop(stack* pin);
void drawDisk(GLUquadricObj** quadric, const GLfloat outer, const GLfloat inner);
void drawPin(GLUquadricObj** quadric, const GLfloat radius, const GLfloat height);
void drawAllPins(GLUquadricObj** quadric, const GLfloat radius, const GLfloat height, const
GLfloat gap);
void drawBitmapString(const GLfloat x, const GLfloat y, const GLfloat z, void* font, char* string);
void drawBitmapInt(const GLfloat x, const GLfloat y, const GLfloat z, void* font, const int number);

void hanoi(actions* queue, const int n, const char pin1, const char pin2, const char pin3)
{
                action* curaction;
                if (n > 0)
                {
                        hanoi(queue, n - 1, pin1, pin3, pin2);
                        /* push action into action queue */
                        curaction = (action*)malloc(sizeof(action));
                        curaction->next = NULL;
                        curaction->fromstack = pin1;
                        curaction->tostack = pin3;
                        if (queue->head == NULL)
                                queue->head = curaction;
                        if (queue->tail != NULL)
                                queue->tail->next = curaction;
                        queue->tail = curaction;
                        hanoi(queue, n - 1, pin2, pin1, pin3);
                }
}

/** push item to pin */
```

```
void push(stack* pin, disk* item) {
		item->next = NULL;
		if (pin->bottom == NULL) {
			pin->bottom = item;
			pin->top = item;
			item->prev = NULL;
		}
		else {
			pin->top->next = item;
			item->prev = pin->top;
			pin->top = item;
		}
}

/** pop item from pin */
disk* pop(stack* pin) {
		disk* tmp;
		if (pin->top != NULL) {
			tmp = pin->top;
			if (pin->top->prev != NULL) {
				pin->top->prev->next = NULL;
				pin->top = tmp->prev;
			}
			else {
				pin->bottom = NULL;
				pin->top = NULL;
			}
			return tmp;
		}
		return NULL;
}

void drawDisk(GLUquadricObj** quadric, const GLfloat outer, const GLfloat inner)
{
		glPushMatrix();
		glRotatef(-90.0, 1.0, 0.0, 0.0);
		gluCylinder(*quadric, outer, outer, BREITE, SLICES, LOOPS);
		gluQuadricOrientation(*quadric, GLU_INSIDE);
		if (inner > 0)
			gluCylinder(*quadric, inner, inner, BREITE, INNERSLICES, LOOPS);
		gluDisk(*quadric, inner, outer, SLICES, LOOPS);
		gluQuadricOrientation(*quadric, GLU_OUTSIDE);
		glTranslatef(0.0, 0.0, BREITE);
		gluDisk(*quadric, inner, outer, SLICES, LOOPS);
		gluQuadricOrientation(*quadric, GLU_OUTSIDE);
		glPopMatrix();
}

void drawPin(GLUquadricObj** quadric, const GLfloat radius, const GLfloat height)
{
		glPushMatrix();
		glRotatef(-90.0, 1.0, 0.0, 0.0);
```

```
		gluCylinder(*quadric, radius, radius, BREITE / 2, SLICES, LOOPS);
		gluQuadricOrientation(*quadric, GLU_INSIDE);
		gluDisk(*quadric, 0.0, radius, SLICES, LOOPS);
		gluQuadricOrientation(*quadric, GLU_OUTSIDE);
		glTranslatef(0.0, 0.0, BREITE / 2);
		gluDisk(*quadric, 0.0, radius, SLICES, LOOPS);
		gluCylinder(*quadric, STANGENBREITE, STANGENBREITE, height,
INNERSLICES, LOOPS);
		glTranslatef(0.0, 0.0, height);
		gluDisk(*quadric, 0.0, STANGENBREITE, INNERSLICES, LOOPS);
		glPopMatrix();
}

void drawAllPins(GLUquadricObj** quadric, const GLfloat radius, const GLfloat height, const
GLfloat gap)
{
		glPushMatrix();
		drawPin(quadric, radius, height);
		glTranslatef(-gap, 0.0, 0.0);
		drawPin(quadric, radius, height);
		glTranslatef(gap * 2, 0.0, 0.0);
		drawPin(quadric, radius, height);
		glPopMatrix();
}

void drawBitmapString(const GLfloat x, const GLfloat y, const GLfloat z, void* font, char* string)
{
		char* c;
		glRasterPos3f(x, y, z);
		for (c = string; *c != '\0'; c++)
			glutBitmapCharacter(font, *c);
}

void drawBitmapInt(const GLfloat x, const GLfloat y, const GLfloat z, void* font, const int number)
{
		char string[17];
		printf(string, "%d", number);
		drawBitmapString(x, y, z, font, string);
}

void populatePin(void)
{
		int i;
		disk* cur;
		GLfloat radius = 0.1f * disks;
		for (i = 0; i < disks; i++)
		{
			cur = (disk*)malloc(sizeof(disk));
			cur->color = (char)i % 6;
			cur->radius = radius;
			push(&pin[0], cur);
			radius -= 0.1;
```

```
		}
		duration = 0;
		draw = 0;
}

void clearPins(void)
{
		int i;
		disk* cur, * tmp;
		free(curdisk);
		curdisk = NULL;
		for (i = 0; i < 3; i++)
		{
			cur = pin[i].top;
			while (cur != NULL)
			{
				tmp = cur->prev;
				free(cur);
				cur = tmp;
			}
			pin[i].top = NULL;
			pin[i].bottom = NULL;
		}
}

void hanoiinit(void)
{
		GLfloat radius;
		speed = FSEM * FEM;
		radius = 0.1f * disks;
		config.pinradius = radius + 0.1f;
		config.gap = radius * 2 + 0.5f;
		config.pinheight = disks * BREITE + 0.2f;
		maxdraws = (2 << (disks - 1)) - 1;   //calculate minimum number of moves
		populatePin();
		/* calculate actions; initialize action list */
		actqueue.head = NULL;
		hanoi(&actqueue, disks, 0, 1, 2);
		curaction = actqueue.head;
		curdisk = pop(&pin[(int)curaction->fromstack]);
		pos = 0.001;
}

void reset(void)
{
		clearPins();
		populatePin();
		/* reset actions */
		curaction = actqueue.head;
		curdisk = pop(&pin[(int)curaction->fromstack]);
		pos = 0.001;
```

```
}

void hanoicleanup(void)
{
		action* acur, * atmp;
		clearPins();
		acur = actqueue.head;
		do {
			atmp = acur->next;
			free(acur);
			acur = atmp;
		} while (acur != NULL);
		gluDeleteQuadric(quadric);
}
void setColor(const int color)
{
		switch (color) {
		case 0:
			glColor3f(1.0, 0.0, 0.0);
			break;
		case 1:
			glColor3f(0.0, 1.0, 0.0);
			break;
		case 2:
			glColor3f(1.0, 1.0, 0.0);
			break;
		case 3:
			glColor3f(0.0, 1.0, 1.0);
			break;
		case 4:
			glColor3f(1.0, 0.0, 1.0);
			break;
		case 5:
			glColor3f(0.0, 0.0, 0.0);
			break;
		}
}

void Init(void)
{
		const GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
		const GLfloat mat_shininess[] = { 50.0 };
		const GLfloat light_position[] = { 0.0, 1.0, 1.0, 0.0 };
		glShadeModel(GL_SMOOTH);
		glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);  /* draw polygons filled */
		glClearColor(1.0, 1.0, 1.0, 1.0);   /* set screen clear color */
		glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);  /* blending
settings */
		glCullFace(GL_BACK);		/* remove backsides */
		glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
		glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
		glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

```
		glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
		glLightModeli(GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE);
		glEnable(GL_LIGHTING);
		glEnable(GL_LIGHT0);
		glEnable(GL_COLOR_MATERIAL);
		glEnable(GL_DEPTH_TEST);
		quadric = gluNewQuadric();
		gluQuadricNormals(quadric, GLU_SMOOTH);
}

/*Is called if the window size is changed */
void Reshape(int width, int height)
{
		glViewport(0, 0, (GLint)width, (GLint)height);
		glMatrixMode(GL_PROJECTION);
		glLoadIdentity();
		gluPerspective(60.0, 1.0, 1.0, 75.0);
		glMatrixMode(GL_MODELVIEW);
}

/* react to key presses */
void Key(unsigned char key, int x, int y)
{
		switch (key)
		{
		case '1':
			disks = 1;
			hanoiinit();
			reset();
			offsetY = 0.9;
			zoom = -1.3;
			gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
			reset();
			break;
		case '2':
			disks = 2;
			hanoiinit();
			reset();
			offsetY = 1.1;
			zoom = -0.8;
			gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
			reset();
			break;
		case '3':
			disks = 3;
			hanoiinit();
			reset();
			offsetY = 1.3;
			zoom = -0.3;
			gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
			break;
		case '4':
```

```
            disks = 4;
            hanoiinit();
            reset();
            offsetY = 1.5;
            zoom = 0.8;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case '5':
            disks = 5;
            hanoiinit();
            reset();
            offsetY = 1.7;
            zoom = 1.3;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case '6':
            disks = 6;
            hanoiinit();
            reset();
            offsetY = 1.9;
            zoom = 1.8;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case '7':
            disks = 7;
            hanoiinit();
            reset();
            offsetY = 2.1;
            zoom = 2.3;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case '8':
            disks = 8;
            hanoiinit();
            reset();
            offsetY = 2.3;
            zoom = 2.8;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case '9':
            disks = 9;
            hanoiinit();
            reset();
            offsetY = 2.5;
            zoom = 3.3;
            gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0);
            break;
        case 27:
        case 'q':
            exit(EXIT_SUCCESS);
            break;
        case ' ':
```

```
                rotX = 0.0;
                rotY = 0.0;
                zoom = 0.0;
                offsetY = 1.5;
                speed = FSEM * FEM;
                break;
        case '+':
                zoom -= 0.1;
                break;
        case '-':
                zoom += 0.1;
                break;
        case 'r':
                reset();
                break;
        case 'f':
                if (fullscreen == 0)
                {
                        glutFullScreen();
                        fullscreen = 1;
                }
                else
                {
                        glutReshapeWindow(800, 600);
                        glutPositionWindow(50, 50);
                        fullscreen = 0;
                }
                break;
        case 's':
                speed += 0.005;
                break;
        case 'x':
                speed -= 0.005;
                if (speed < 0.0)
                        speed = 0.0;
                break;
        }
        glutPostRedisplay();
}

void SpecialKey(int key, int x, int y)
{
        switch (key)
        {
        case GLUT_KEY_UP:
                rotX -= 5;
                break;
        case GLUT_KEY_DOWN:
                rotX += 5;
                break;
        case GLUT_KEY_LEFT:
                rotY -= 5;
```

```
                break;
        case GLUT_KEY_RIGHT:
                rotY += 5;
                break;
        case GLUT_KEY_PAGE_UP:
                offsetY -= 0.1;
                break;
        case GLUT_KEY_PAGE_DOWN:
                offsetY += 0.1;
                break;
        }
        glutPostRedisplay();
}

void
mouse(int button, int state, int x, int y)
{

        if (button == GLUT_LEFT_BUTTON && state == GLUT_DOWN)
        {
                zoom += 0.1;
        }
        if (button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
                zoom -= 0.1;
        if (button == GLUT_MIDDLE_BUTTON && state == GLUT_DOWN)
                reset();


}

void Display(void)
{
        disk* cur;
        int i;
        GLfloat movY;
        glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT); /* clear scren */
        glLoadIdentity();
        glColor3f(0.0, 0.0, 0.0);
        drawBitmapString(25.0, 32.0, -60.0, GLUT_BITMAP_9_BY_15, seconds);
        drawBitmapInt(25.0, 30.0, -60.0, GLUT_BITMAP_9_BY_15, draw);
        drawBitmapInt(28.0, 30.0, -60.0, GLUT_BITMAP_9_BY_15, maxdraws);

        const char* s1 = "Controls:";
        char* p1 = (char*)s1;
        drawBitmapString(-32.0, 32.0, -60.0, GLUT_BITMAP_9_BY_15, p1);

        const char* s2 = "s : increase speed";
        char* p2 = (char*)s2;
        drawBitmapString(-32.0, 30.0, -60.0, GLUT_BITMAP_9_BY_15, p2);

        const char* s3 = "x : decrease speed";
        char* p3 = (char*)s3;
```

```
drawBitmapString(-32.0, 28.0, -60.0, GLUT_BITMAP_9_BY_15, p3);

const char* s4 = "- : zoom out";
char* p4 = (char*)s4;
drawBitmapString(-32.0, 26.0, -60.0, GLUT_BITMAP_9_BY_15, p4);

const char* s5 = "+ : zoom in";
char* p5 = (char*)s5;
drawBitmapString(-32.0, 24.0, -60.0, GLUT_BITMAP_9_BY_15,p5 );

const char* s6 = "f : fullscreen";
char* p6 = (char*)s6;
drawBitmapString(-32.0, 22.0, -60.0, GLUT_BITMAP_9_BY_15,p6 );

const char* s7 = "r : restart animation";
char* p7 = (char*)s7;
drawBitmapString(-32.0, 20.0, -60.0, GLUT_BITMAP_9_BY_15,p7);

const char* s8 = "f : fullscreen";
char* p8 = (char*)s8;
drawBitmapString(-32.0, 18.0, -60.0, GLUT_BITMAP_9_BY_15,p8);

const char* s9 = "q/Esc : quit";
char* p9 = (char*)s9;
drawBitmapString(-32.0, 16.0, -60.0, GLUT_BITMAP_9_BY_15, p9);


const char* s10 = "Use the arrow keys to rotate the view";
char* p10 = (char*)s10;
drawBitmapString(-32.0, 14.0, -60.0, GLUT_BITMAP_9_BY_15, p10);

const char* s11 = "Use PageUp/PageDown to move the view up/down";
char* p11 = (char*)s11;
drawBitmapString(-32.0, 12.0, -60.0, GLUT_BITMAP_9_BY_15, p11);

const char* s12 = "Press Spacebar to reset the view";
char* p12 = (char*)s12;
drawBitmapString(-32.0, 10.0, -60.0, GLUT_BITMAP_9_BY_15, p12);

const char* s13 = "Press 1 - 9 to select the number of disks";
char* p13 = (char*)s13;


drawBitmapString(-32.0, 8.0, -60.0, GLUT_BITMAP_9_BY_15,p13);
gluLookAt(0.0, 0.9, 3.6 + zoom, 0.0, offsetY, 0.0, 0.0, 1.0, 0.0); // calculate view
point
glRotatef(rotY, 0.0, 1.0, 0.0);          /* rotate Y axis */
glRotatef(rotX, 1.0, 0.0, 0.0);          /* rotate X axis */
glColor3f(0.0, 0.0, 0.5);
drawAllPins(&quadric, config.pinradius, config.pinheight, config.gap); // draw pins
glTranslatef(-config.gap, BREITE / 2, 0.0);
glPushMatrix();
```

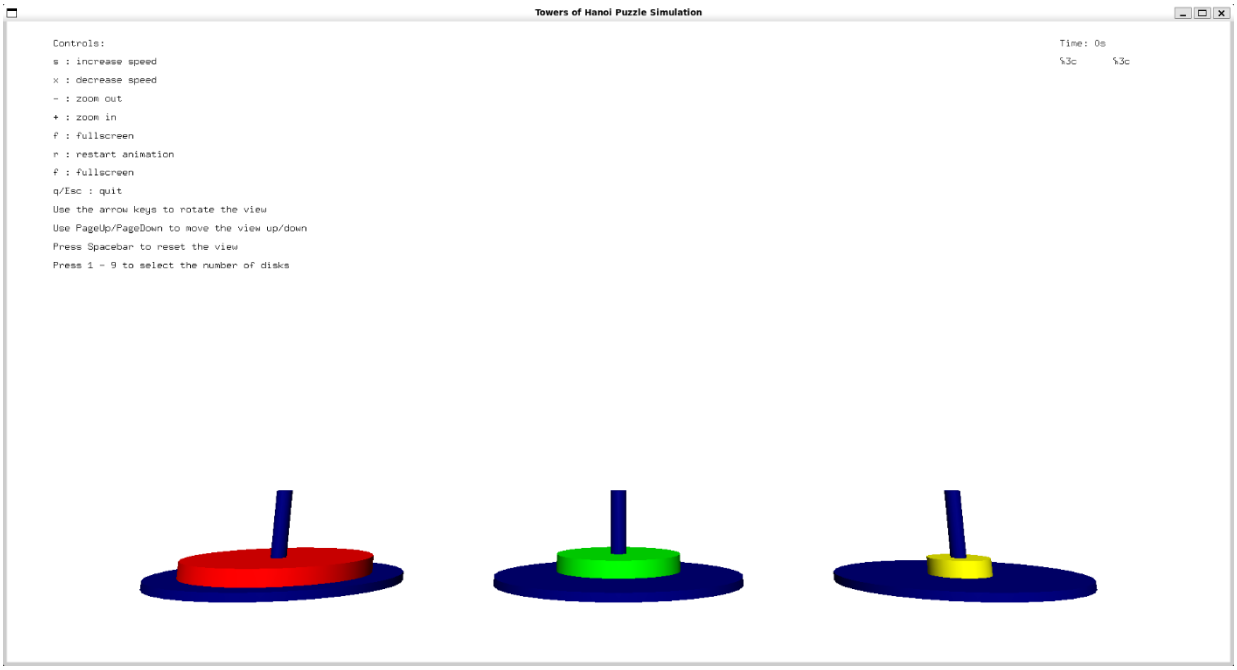
```
		for (i = 0; i < 3; i++)
		{ /* fill pins with disks */
			glPushMatrix();
			pinheight[i] = 0;
			if ((cur = pin[i].bottom) != NULL)
			{
				do {
					setColor(cur->color);
					drawDisk(&quadric, cur->radius, STANGENBREITE);
					glTranslatef(0.0, BREITE, 0.0);
					pinheight[i] += BREITE;
					cur = cur->next;
				} while (cur != NULL);
			}
			glPopMatrix();
			glTranslatef(config.gap, 0.0, 0.0);
		}
		glPopMatrix();
		if (curaction != NULL && curaction->fromstack != -1 && curdisk != NULL) {
			if (pos <= 1.0)
			{
				movY = pos * (config.pinheight - pinheight[(int)curaction-
>fromstack]);
				glTranslatef(config.gap * curaction->fromstack,
pinheight[(int)curaction->fromstack] + movY, 0.0);
			}
			else
			{
				if (pos < 2.0 && curaction->fromstack != curaction->tostack)
				{
					if (curaction->fromstack != 1 && curaction->tostack != 1)
					{ /* jump 2 pins */
						glTranslatef(config.gap, config.pinheight + 0.05f, 0.0);
						if (curaction->fromstack == 0)
							glRotatef(-(pos - 2.0f) * 180 - 90, 0.0, 0.0, 1.0);
						else
							glRotatef((pos - 2.0f) * 180 + 90, 0.0, 0.0, 1.0);
						glTranslatef(0.0, config.gap, 0.0);
					}
					else
					{
						if (curaction->fromstack == 0 && curaction->tostack ==
1)
						{
							glTranslatef(config.gap / 2, config.pinheight +
0.05f, 0.0);
							glRotatef(-(pos - 2.0f) * 180 - 90, 0.0, 0.0, 1.0);
						}
						else
						{
							if (curaction->fromstack == 2 && curaction-
>tostack == 1)
```

```
{
glTranslatef(config.gap / 2 * 3, config.pinheight + 0.05f, 0.0);
glRotatef((pos - 2.0f) * 180 + 90, 0.0, 0.0, 1.0);
                }
else
{
if (curaction->fromstack == 1 && curaction->tostack == 2)
{
glTranslatef(config.gap / 2 * 3, config.pinheight + 0.05f, 0.0);
glRotatef(-(pos - 2.0f) * 180 - 90, 0.0, 0.0, 1.0);
                                                        }
else{
                glTranslatef(config.gap / 2, config.pinheight + 0.05f, 0.0);
                glRotatef((pos - 2.0f) * 180 + 90, 0.0, 0.0, 1.0);
                }
    }
}
glTranslatef(0.0, config.gap / 2, 0.0);
}
glRotatef(-90, 0.0, 0.0, 1.0);
}
else
if (pos >= 2.0)
{ /* drop disk down */
movY = config.pinheight - (pos - 2.0f + speed) * (config.pinheight - pinheight[(int)curaction
>tostack]);
glTranslatef(config.gap * curaction->tostack, movY, 0.0);
                                        }
                }
                setColor(curdisk->color);
                drawDisk(&quadric, curdisk->radius, STANGENBREITE);
        }
        glutSwapBuffers(); /* swap buffers (double buffering) */
}
void moveDisk(int param)
{
        if (param == 1)
                reset();
        if (curaction != NULL)
        {
                if (pos == 0.0 || pos >= 3.0 - speed)
                { /* 0--1 -> disk goes upwards, 1--2 "disk in air", 2--3 disk goes downwards*/
                        pos = 0.0;
                        draw++;
                        push(&pin[(int)curaction->tostack], curdisk);
                        curaction = curaction->next;
                        if (curaction != NULL)
                                curdisk = pop(&pin[(int)curaction->fromstack]);
                }
                pos += speed;
                if (pos > 3.0 - FSEM)
                        pos = 3.0 - FSEM;
```

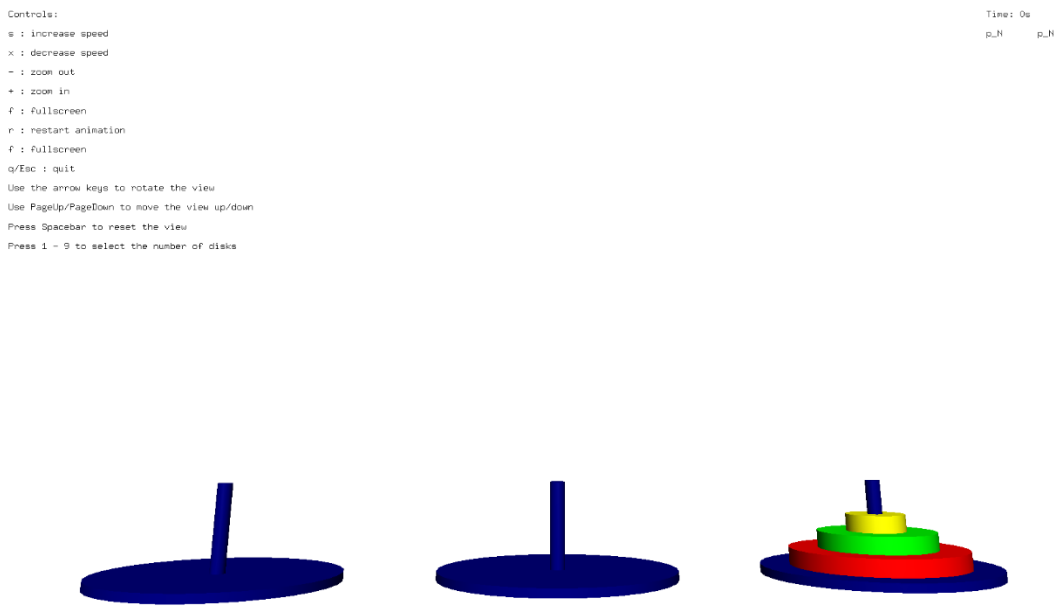
```
                glutTimerFunc((unsigned)FEM, moveDisk, 0);
        }
        else
        {
                curdisk = NULL;
                glutTimerFunc(5000, moveDisk, 1);
        }
        glutPostRedisplay();
}

void timer(int param)
{
        if (curaction != NULL)
        {
                printf(seconds, "Time: %ds", ++duration);
        }
        glutTimerFunc(1000, timer, 0);
}

int main(int argc, char* argv[])
{
        hanoiinit();
        atexit(hanoicleanup);
        glutInit(&argc, argv);
        //command line arguments for setting the number of disks
        if (argc > 1)
        glutInitWindowPosition(0, 0);
        glutInitWindowSize(800, 600);
        glutInitDisplayMode(GLUT_DEPTH | GLUT_DOUBLE);
        if (glutCreateWindow("Towers of Hanoi Puzzle Simulation") == GL_FALSE)
                exit(EXIT_FAILURE);
        Init();
        glutReshapeFunc(Reshape);
        glutKeyboardFunc(Key);
        glutSpecialFunc(SpecialKey);
        glutMouseFunc(mouse);
        glutDisplayFunc(Display);
        glutTimerFunc((unsigned)FEM, moveDisk, 0);
        glutTimerFunc(1000, timer, 0);
        glutMainLoop();
        return EXIT_SUCCESS;
}
```

Chapter-4

Results



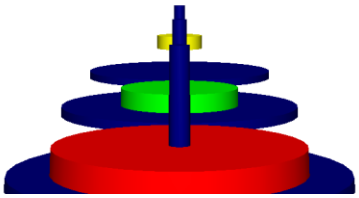
FRONT VIEW



BACK VIEW

Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks

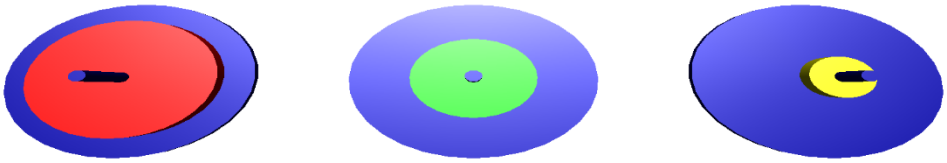
Time: 0s



LEFT VIEW

Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks

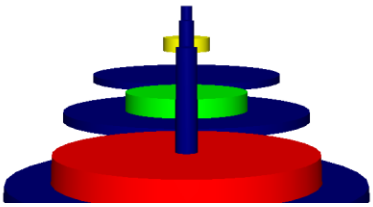
Time: 0s



TOP VIEW

Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks

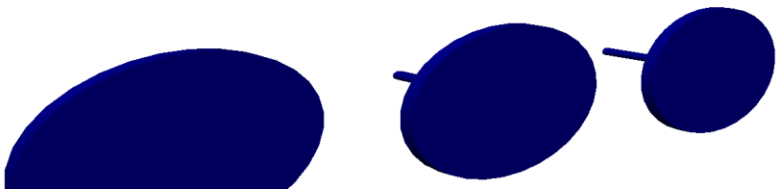
Time: 0s



RIGHT VIEW

Controls:
s : increase speed
x : decrease speed
- : zoom out
+ : zoom in
f : fullscreen
r : restart animation
f : fullscreen
q/Esc : quit
Use the arrow keys to rotate the view
Use PageUp/PageDown to move the view up/down
Press Spacebar to reset the view
Press 1 - 9 to select the number of disks

Time: 0s



BOTTOM VIEW

Chapter-5

Conclusion

In conclusion, the Tower of Hanoi is a fascinating mathematical puzzle that challenges our problem-solving skills and logical reasoning. The puzzle consists of three rods and a set of different-sized disks, with the objective of moving all the disks from one rod to another while following specific rules.

Throughout the solution process, several key observations become apparent. First, as the number of disks increases, the complexity of the puzzle grows exponentially. The minimum number of moves required to solve the Tower of Hanoi with n disks is 2^n - 1, which highlights the exponential nature of the problem.

Additionally, the Tower of Hanoi puzzle demonstrates the concept of recursion and the power of dividing a complex problem into smaller, manageable subproblems. By breaking down the problem into smaller towers and applying the same set of rules, we can solve each subproblem independently and then combine the solutions to solve the larger problem.

The Tower of Hanoi also serves as a metaphor for various real-life scenarios. It illustrates the importance of planning, patience, and strategic thinking in problem-solving. The puzzle teaches us that taking a step back, analyzing the situation, and considering different approaches can help us find efficient solutions to complex problems.

Furthermore, the Tower of Hanoi puzzle has applications in computer science and algorithm design. It is often used as an example to explain the concept of recursion and to analyze the efficiency of algorithms. The puzzle's simple yet challenging nature makes it an ideal problem for teaching and learning fundamental programming concepts.

In conclusion, the Tower of Hanoi is not only an entertaining puzzle but also a valuable tool for developing problem-solving skills, understanding recursion, and exploring algorithmic concepts. Its timeless appeal continues to captivate enthusiasts and serves as a testament to the beauty and elegance of mathematics and logical thinking.

Chapter-6

Bibliography

1. James D Foley, Andreis Van Dam, Steven K Finer, John F Huges Computer graphics with OpenGL: Pearson education
2. Official OPENGL Documentation at

   https://www.opengl.org/Documentation/Specs.html
3. Youtube.com