MAHAWAR IN OPENGL

Enrolment no. -081268

Name of student-Anshul Gupta

Enrolment no. -081274

Name of Student-Jitendra Kalwaniyan

Name of Supervisor-Suman Saha

MAY 2012

Project Report submitted in partial fulfillment of the requirement for the degree of

Bachelor of Technology

Department of

COMPUTER SCIENCE AND ENGINEERING

Jaypee University of Information and Technology

Waknaghat, Solan – 173234, Himachal Pradesh

TABLE OF CONTENTS

Chapter No.	Topics	Page No.
	INNER FIRST PAGE	**1**
	TABLE OF CONTENTS	**2-4**
	CERTIFICATE	**5**
	ACKNOWLEDGEMENT	**6**
	PREFACE	**7**
	ABSTRACT	**8**
	LIST OF ABBREVIATIONS	**9**
	LIST OF FIGURES	**10**
CHAPTER 1	Introduction	11
CHAPTER 2	Game Background	12
CHAPTER 3	Game Rules	13
	3.1 How to play	13-14
	3.2 About the game	14-15
	3.3 Pawn Movements	15-18
CHAPTER 4	About OPENGL	19-32
CHAPTER 5	5 Analyses and Design	33-35
	5.1 GUI	33
	5.2 INPUT and Controls	33

5.3 View of the game board…………………………………………...34

5.3.1 Orientation……………………………………………………34

5.3.2 Texture……………………………………………………..34

5.4 Pawns……………………………………………………...34

5.5 Treasure Keys…………………………………………………..34

5.6 Treasure Maps………………………………………………….34

5.7 Dice………………………………………………………..35

CHAPTER 6 Development Stages…………………………………………………..36

CHAPTER 7 Module specification…………………………………………………..37

7.1 Game Board…………………………………………………37

7.2 Pawns…………………………………………………..37-38

7.3 Dice…………………………………………………………..38

7.4 Treasure Keys…………………………………………………..38

7.5 Treasure Maps………………………………………………….38

CHAPTER 8 Testing and Debugging…………………………………………………..39

CHAPTER 9 Future Development and Extensions……………………………………….40

CHAPTER 10 Limitations……………………………………………………………….41

CHAPTER 11 Division of work………………………………………………………...42

CHAPTER 12 Conclusion……………………………………………………………….43

CHAPTER 13 Appendix………………………………………………………………….44

A. Code ……………………………………………………………………..44-55

B. References………………………………………………………………………...56

CERTIFICATE

This is to certify that project report entitled "MAHAWAR IN OPENGL", submitted by Anshul Gupta (081268) and Jitendra Kalwaniyan (081274) in partial fulfillment for the award of degree of Bachelor of Technology in Computer Science and Engineering to Jaypee University of Information Technology, Waknaghat, Solan has been carried out under my supervision.

This work has not been submitted partially or fully to any other University or Institute for the award of this or any other degree or diploma.

Date: [Mr. Suman Saha]

Senior Lecturer (Department of CSE and IT)

ACKNOWLEDGEMENT

This project is an outcome of our serious effort to simulate board game known as "MAHAWAR" in software using OPENGL API as part of our Bachelor's Degree Program.

Software simulation of the board game involves the concept of computer graphics, 3D rendering and C language; under guidance our esteemed mentor Senior Lecturer **Mrs. Meenakshi Arya**, **Ms. Madhu Kumari** and **Mr. Suman Saha** not only cleared all our ambiguities but also generated a high level of interest in the subject. We are highly grateful to them.

The prospect of working in a group with a high level of accountability fostered a spirit of teamwork and created a feeling of oneness which thus motivated us to perform to the best our ability and create a report of the highest quality.

To do only the best quality work, with utmost sincerity and precision has been our constant endeavor.

This work has not been submitted partially or wholly to any other University or Institute for the award of this or any other degree or diploma.

Date: [Anshul Gupta (081268)]

[Jitendra Kalwaniyan (081274)]

Preface

One is probably familiar with the exceptional benefits of playing a game. Regardless of one's age and physical ability, games such as golf, billiards, board games, etc. are considered to be those types of recreational activities that people select to try in order to increase their mental and/or physical skills while enjoying the excitement. Games as a result have always been an inseparable part of human existence, especially board games.

In our era of endless distractions, it's easy to forget how important board games to our ancestors. There is possibly no entertainment for such a huge period of human existence. In all environments, board games have a fantastically strong hold. They reigned supreme. For centuries, even millenniums, board games have served as the play stations of the early days.

A decade ago, Indian kids grew up playing board games, ones which needed logical thinking. However, with the advent of various multimedia games, the traditional board game has taken a backseat. Board games play a vital role in the emotional and social development of a child. It helps improve their imagination and communication skills. Games need not be academic to be educational. It is these games that help children learn social etiquette; taking turns, being patient and teamwork to name a few.

Chess has been the only version of software simulated board games in our present gaming format. With a view to broaden the extent of software simulated board games a sincere effort has been made by us.

We are thankful to all those people who have been directly or indirectly given there crucial support and precious time for the successful completion of the project.
First and foremost we take this opportunity to acknowledge the support rendered by our project guides: **Mrs. Meenakshi Arya**, **Ms. Madhu Kumari** and our current project guide **Mr. Suman Saha**.We profusely thank our lab Coordinators and friends who worked hard and given their kind cooperation for the entire duration of the project and until the very end.

Anshul Gupta
Jitendra Kalwniyan

Abstract

Mahawar is derived from the game of **Chess** and **Chaupad** (which is a board game of cross and circles). Both of these games are believed to have been developed during 6th century and 4th century respectively. There are palaces in Allahabad and Agra which served as giant Chaupad boards for the Indian Emperor Akbar I from the Mogul Empire in the 16th century. The board was made of inlaid marble and it had red and white squares. He used to sit in the center and toss the shells; 16 women from his harem were the pawns and they moved the way he told them. Chess is commonly believed to have originated in northwest India during the Gupta empire, where its early form in the 6th century was known as *chaturaṅga* (Sanskrit: four divisions [of the military] – infantry, cavalry, elephants, and chariotry, represented by the pieces that would evolve into the modern pawn, knight, bishop, and rook, respectively). The earliest evidence of chess is found in the neighboring Sassanid Persia around 600, where the game came to be known by the name *chatrang*. Chatrang is evoked in three epic romances written in Pahlavi (Middle Persian).

This project is a software simulation of a board game Known as **Mahawar** designed and developed by an IIT Bombay student **Kumar Ahir**. Opengl API has been used for the development of the project due to its ability for faster draw calls and ease in rendering complex objects.

List of Figures

Figure 1	**An image rendered using opengl**
Figure 2	**Opengl rendering pipeline**
Figure 3	**Learning to shade in Opengl**
Figure 4	**Implementing Scaling**
Figure 5	**Implementing key movement and color change and zooming**
Figure 6	**Game Board**
Figure 7	**Dice**

List of Abbreviations

Opengl	**Open Graphics Library**
gl	**Graphics Library**
glu	**Graphics Library Utility**
	Graphics Library Utility Toolkit
RGB	**Red, Green, Blue**
RGBA	**Red, Green, Blue, Alpha**
GLSL	**Graphics Library Shading Language**
GLEW	**Graphics Library Extension Wrangler**
AGL	**Opengl Extension for the Apple Macintosh**
WGL	**Opengl Extension for Microsoft Windows**

Introduction

With the advent of computers the more conventional physical format of the games has become less common among the youngsters. The steep inclination in the popularity of software games is tremendous. The release of software games is sometimes possibly more awaited than there school results. Games such as **counter strike**, **need for speed, grand theft auto** and more recent games such as **mass effect**, **batman, L.A. noire** have become more than addiction to the game lovers.

Fascinated and intrigued by the complexity of the games from an engineering student's perspective we decided to take a small game as a part of our final year project.

Our first task was to select the game that we wanted to work upon. A game that is neither too complex nor too small so that we are able to not only learn enough and effectively but also finish it on time. Our project guide helped us solve our dilemma. We chose a board game known as MAHAWAR developed by an IIT student Kumar Ahir. Then the decision was to be made regarding the technology that we wanted to choose for simulation of our game. After having a deeper analysis of some of the most popular PC games we found out a very interesting fact that most of them are developed using a similar technology and that was OPENGL. So we decided to continue our project in the same technology.

Game Background

Mahawar is a game developed by a student of IIT Bombay primarily derived from some of the most popular board games like chess, Chaupad and pachisi. Each of these games had been played in India from the very beginning of their introduction i.e. before 4th century.

Chess is commonly believed to have originated in northwest India during the Gupta empire, where its early form in the 6th century was known as *chaturaṅga* (Sanskrit: four divisions [of the military] – infantry, cavalry, elephants, and chariotry, represented by the pieces that would evolve into the modern pawn, knight, bishop, and rook, respectively). The earliest evidence of chess is found in the neighboring Sassanid Persia around 600, where the game came to be known by the name *chatrang*. Chatrang is evoked in three epic romances written in Pahlavi (Middle Persian). Chatrang was taken up by the Muslim world after the Islamic conquest of Persia (633–644), where it was then named *shatranj*, with the pieces largely retaining their Persian names. In Spanish "shatranj" was rendered as *ajedrez* ("al-shatranj"), in Portuguese as *xadrez*, and in Greek as ζατρίκιον (*zatrikion*, which comes directly from the Persian *chatrang*), but in the rest of Europe it was replaced by versions of the Persian *shāh* ("king"), which was familiar as an exclamation and became the English words "check" and "chess". Murray theorized that Muslim traders came to European seaports with ornamental chess kings as curios before they brought the game of chess.

Chaupad is a board game of the Cross and Circle family played in India that is very similar to Pachisi. It is believed that both games were created around the 4th century. The board is made of wool or cloth. The dice are six cowry shells and the pawns are made of wood. It's usually played on a table or the floor.

There are palaces in Allahabad and Agra which served as giant Chaupad boards for the Indian Emperor Akbar I from the Mogul Empire in the 16th century. The board was made of inlaid marble and it had red and white squares. He used to sit in the center and toss the shells; 16 women from his harem were the pawns and they moved the way he told them.

Game rules

Ages – 8 years +

No. of players – 2 to 4

Contents:

1. Battle zones
2. 12 treasure Map Cards

Game Summary:

This game features a game board with four player battle zones and treasure zones being in the midst of the battle zones. Strategically deploy your unique set of skilled warriors, move them towards the treasure zone and get clues to find out which king has the right key to the treasure chest. Proceed to that king's castle and fight for the key, come back to the treasure zone and unlock the treasure chest with the Key. The king who first brings the right key to the treasure zone will be the winner.

Setting up the game:

The game demands a careful planning before you actually start the play. The initial placement of the warrior pawns on your battle zone will determine your strategy in gameplay. Therefore it is important to understand the significance of each warrior pawn by carefully reading the instructions carefully.

Each player's battle zone has 36 square blocks. The players then have to do the following on their respective battle zone.

A.) Placement of life and death resource pawns:

Each player gets two resource pawns (one LIFE and one DEATH). The player must first place his/her resource pawns on two blocks anywhere in their respective player zones. The resource pawns should however be placed such that the diagonals passing through them don't intersect each other in any block. The pawns are placed on the game board such that the opponent players will not be able to recognize whether it is a life pawn or a death pawn i.e. the bottom view faces the game board.

B.) Placement of warrior pawns

Each player chooses his/her set of warrior pawns. The warrior pawns consist of the following Chakra, Fireball, Spear, Shield, Axe, Archer and Horse. At the beginning of the game the player

places the shield warrior pawn on the top of the spear pawn to form the combined Spear and Shield pawn which is then considered as one warrior pawn during gameplay.

These two warriors Shield and Spear always fight together and are considered as one warrior during the battle, but the added advantage is that they have two lives. So if the combined Spear and Shield warrior is killed by an enemy once then the warrior uses his extra lives and continues to fight in the battle. Upon the first killing the Shield pawn on top of the spear pawn is removed from the play.

The players then places then place their warriors and resource pawns in any of the 36 blocks within their territory battle zone; however they need to observe the following rules:

1. Each player can place only one warrior pawn in the first row (Orange) of his battle zone (the row adjacent facing the treasure zone).
2. In the second row (Blue), the player can place a maximum of two warrior pawns in the battle zone.
3. The player can place the remaining warrior pawns in any square block from row numbered 3 onwards.
4. At the beginning of the game no two warriors can occupy the same square block in the player 's battle zone with the exception of the Spear and Shield Warrior pawns during placement of Warrior pawns.

C.)Placement of treasure keys:

Each player also gets 1 treasure key which needs to be placed in the blocks where the castle of the respective player lies.

D.) Forming of treasure map and treasure zone

There are 12 treasure map cards in the game. They consist of the following:

1.) 4 nos. of treasure found 4 treasure map cards indicating which king has the right key to the treasure: For example: "TREASURE FOUND: key with king 2".
2.) 8 nos. of condition cards – 7 treasure map cards have certain conditions which have to be met when a player lands on these cards in the treasure zone and one treasure map card is blank. The player does nothing when landing on this card in the treasure zone.

At the beginning of each game, take the 4 "Treasure Found" map cards which indicates which king has the right key to the treasure, shuffle them and allow any other player to select

1 card (without seeing the content). The "Treasure Found selected card" is to be mixed along with 8 balance condition cards and will be placed in the center of the game board kept face down

to form the treasure zone. Keep aside the other three ”Treasure Found” cards and do not use them in the game at all.

Forming the treasure zone

A treasure map is prepared to guide players through the treasure zone. Each treasure map card is a 2x2 grid consisting of 4 blocks. 9 treasure cards placed in the center form the treasure zone.

Significance of pawns

A.) Significance of life and death resource pawns:
LIFE: when any warrior pawn of a player lands on the block (in the enemy zone) having the life (resource) a life is rewarded to the player. The player can bring back any player that has been killed earlier. Incase no warrior pawns have been killed earlier his life saving chance can be used later in the game when the player loses a warrior pawn.
DEATH: When a warrior pawn of the player lands on the block on the enemy zone having the death (resource) the pawn is considered dead and has to be removed from the game board.

B.) Significance of Warrior pawns:
It is important that before each battle the player understand the significance of each warrior pawn and the role that they play in the battle. The warrior pawns are of different heights to demonstrate their ferocity in the battle. The tallest warrior (Fireball) is the most lethal, whereas the shortest (Horse) is the weakest.

Fireball:

Directions: these warriors can move only in horizontal and vertical straight line.

Movements: When the player rolls the dice, he/she can move the fireball pawn the number of spaces rolled on the dice in a horizontal or vertical manner. The pawn cannot move if any other warrior pawn is on its path.

Attack strategy of fireball: The Fireball can be used in two ways:

a. Move and kill: Here the players use the fireball to unleash terror and kill all warriors (including his own team warriors) by doing the following step by step:
1. The player needs to call out “kill” and inform all other players of his intention to kill before he moves his pawn.
2. After moving his pawn all the warrior pawns (including his own) that lie in blocks adjacent (not diagonal) to the block where the Fireball will finally land on, will be considered kill. Please remove the dead warriors from the game board.

3. The Fireball is then immediately placed back in an empty block adjacent to his own castle in order to get reloaded. If both the blocks adjacent to the castle is occupied by other warrior/resource pawns, the player must then place the Fireball in any other block in the row where the castle lies.
4. Fireball pawn cannot kill unless reloaded. In the rare possibility, that all the blocks in the 6th row in which the castle lies is occupied, the fireball must wait for ann empty block to get reloaded.

b. Pure move and no kill: In a 'pure move and no kill', the player no intention to kill any other pawn. He may use this strategy to find and open the treasure map cards, when he lands on the treasure zone.

CHAKRA

Directions: These warrior pawns can move only in diagonal line with no turn.

Movements: When the player rolls the dice, he can move the chakra pawn only in a diagonal manner. The chakra pawn has a special feature where it can change direction once during every turn (either to the left or right depending upon the player's choice). The change in direction either to the left or right can happen after any number of blocks, the number however must be less than the number rolled on the dice. For example if a player rolls a number 4 on the dice the player can move his chakra pawn 3 blocks ahead in the diagonal manner and then to the left/right block or the player can decide to move only 2 spaces in the diagonal manner and then turn to the left/right block. The special feature of the chakra pawn is that it can move over any warrior pawn that lies in its path without killing the pawns.

Attack strategy of chakra: If any enemy's is on the block where the chakra finally lands, that enemy warrior pawn is killed and is removed from the game board.

SPEAR AND SHIELD

At the start of the game, each player gets spear and shield as two separate pieces. The player places the shield pawn on the top of the spear pawn to form a combined spear and shield pawn.

Directions: This pawn can move only in horizontal or vertical direction.

Movements: When the player rolls the dice, he/she move the spear and shield pawn the number of the spaces rolled on the dice. the combined pawn cannot move if any other pawn is in the path.

Attack strategy of spear and shield: If any enemy pawn is located on the block where the combined pawn move to after the dice rolled, that enemy warrior pawn is considered killed. Since the combines pawn is combination of two pawn so it has to be attacked twice for elimination.

ARCHER

Direction: This pawn can only move in diagonal line.

Attack strategy of chakra: If any enemy's is on the block where the chakra finally lands ,that enemy warrior pawn is killed and is removed from the game board

AXE

Direction: This pawn can move only in horizontal or vertical direction.

Movements: When the player rolls the dice, he/she move the axe pawn the number of the spaces rolled on the dice. The pawn cannot move if any other pawn is in the path .

Attack strategy of Axe: if any enemy pawn is located on the block where the axe pawn move to after the dice rolled, that enemy warrior pawn is considered killed.

HORSE

Direction: This pawn can move only in horizontal or vertical direction with one turn.

Movements: When the player rolls the dice, he/she move the horse pawn the number of the spaces rolled on the dice with the one turn. The pawn cannot move if any other pawn is in the path .the horse pawn can choose any direction just before the last move. it can also land to a block where the warrior pawn of same team is standing

Attack strategy of Horse: The horse pawn has no power to kill.

Special powers of horse when combined with other pawns:

Horse + chakra:

Movement: Horizontal, vertical and diagonal with one turn, cannot move over pawn in its path.

Action: Move to destination block as per the roll of dice and kills enemy's pawn.

Horse + Fireball:

Movement: Horizontal and vertical with one turn cannot move over pawn in its path.

Action: if horse moves with fireball to kill, the horse also get killed, fireball returns back to castle for reloading.

Horse + Spear Shield:

Movement: Horizontal and vertical with one turn, cannot move over pawn in its path

Action: Move to destination blocks and kill the enemy.

Horse + Spear (after shield pawn killed):

Movement: Horizontal and vertical with one turn, cannot move over pawn in its path.

Action: move to destination block and kill the enemy.

Horse +Archer:

Movement: Horizontal, vertical and diagonal with one turn, cannot move over pawn in its path.

Action: Stay at same block and kills pawn diagonally

Horse + Axe man:

Movement: Horizontal and vertical with one turn, cannot move over pawn in its path.

Action: Move to destination blocks and kill the enemy.

Reading the Treasure Map (During Gameplay): When any warrior pawn lands on a treasure zone block the player then picks up the respective treasure map card, if the card picked reads "Treasure Found: Key with the king 1" the player only says "Treasure Found" to all the other players and does not disclose any other information. If the card picked up has certain other condition then the player reads it aloud and exercises it accordingly.

Warrior and Key:

Once the player knows which king has the treasure key to the right treasure, he can then proceed by moving any of the warrior pawns and attack the king. Upon reaching that castle the warrior then has the possession of the treasure key on the small hole provided on the pawn.

Transfer of Treasure keys between opposing warriors:

If a warrior pawn carrying the treasure key is attacked and killed by any enemy warrior pawn, the treasure is then handed over to that warrior pawn.

About OPENGL

The OpenGL graphics system is a software interface to graphics hardware. “GL” stands for “Graphics Library”. It allows you to create interactive programs that produce color images of moving, three-dimensional objects. With OpenGL, you can control computer-graphics technology to produce realistic pictures, or ones that depart from reality in imaginative ways. This guide explains how to program with the OpenGL graphics system to deliver the visual effect you want.

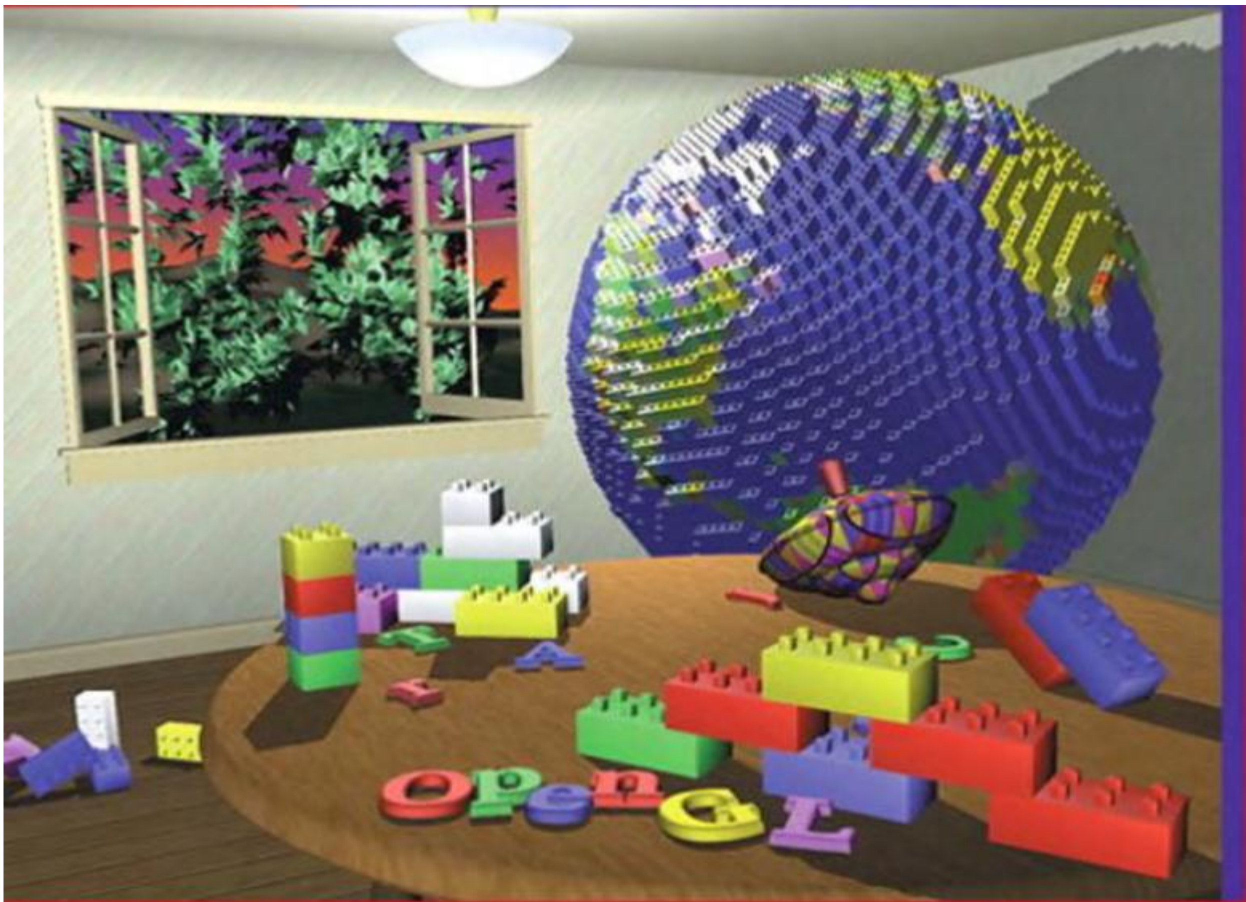

Figure 1: An Image rendered using opengl

The following list describes in general terms how these pictures were made.

• Plate 1 shows the entire scene displayed as a *wireframe* model—that is, as if all the objects in the scene were made of wire. Each *line* of wire corresponds to an edge of a primitive (typically a polygon). For example, the surface of the table is constructed from triangular polygons that are positioned like slices of pie.

Note that you can see portions of objects that would be obscured if the objects were solid rather than wireframe. For example, you can see the entire model of the hills outside the window even though most of this model is normally hidden by the wall of the room. The globe appears to be nearly solid because it's composed of hundreds of colored blocks, and you see the wireframe lines for all the edges of all the blocks, even those forming the back side of the globe. The way the globe is constructed gives you an idea of how complex objects can be created by assembling lower-level objects.

• Plate 2 shows a *depth-cued* version of the same wireframe scene. Note that the lines farther from the eye are dimmer, just as they would be in real life, thereby giving a visual cue of *depth*. OpenGL uses atmospheric effects (collectively referred to as *fog*) to achieve depth cueing.

• Plate 3 shows an *antialiased* version of the wireframe scene. Antialiasing is a technique for reducing the jagged edges (also known as *jaggies*) created when approximating smooth edges using *pixels*—short for *picture elements*—which are confined to a rectangular grid. Such jaggies are usually the most visible, with near-horizontal or near-vertical lines.

• Plate 4 shows a *flat-shaded*, *unlit* version of the scene. The objects in the scene are now shown as solid. They appear "flat" in the sense that only one color is used to render each polygon, so they don't appear smoothly rounded. There are no effects from any light sources.

• Plate 5 shows a *lit, smooth-shaded* version of the scene. Note how the scene looks much more realistic and three-dimensional when the objects are shaded to respond to the light sources in the room, as if the objects were smoothly rounded.

• Plate 6 adds *shadows* and *textures* to the previous version of the scene. Shadows aren't an explicitly defined feature of OpenGL (there is no "shadow command"), allows you to apply a two-dimensional image onto a three-dimensional object. In this scene, the top on the table surface is the most vibrant example of texture mapping. The wood grain on the floor and table surface is all texture mapped, as well as the wallpaper and the toy top (on the table).

• Plate 7 shows a *motion-blurred* object in the scene. The sphinx (or dog, depending on your Rorschach tendencies) appears to be captured moving forward, leaving a blurred trace of its path of motion.

• Plate 8 shows the scene as it was drawn for the cover of the book from a different viewpoint. This plate illustrates that the image really is a snapshot of models of three-dimensional objects.

• Plate 9 brings back the use of fog, which was shown in Plate 2 to simulate the presence of smoke particles in the air. Note how the same effect in Plate 2 now has a more dramatic impact in Plate 9.

• Plate 10 shows the *depth-of-field effect*, which simulates the inability of
a camera lens to maintain all objects in a photographed scene in focus. The camera focuses on a particular spot in the scene. Objects that are significantly closer or farther than that spot are somewhat blurred.

OpenGL Rendering Pipeline

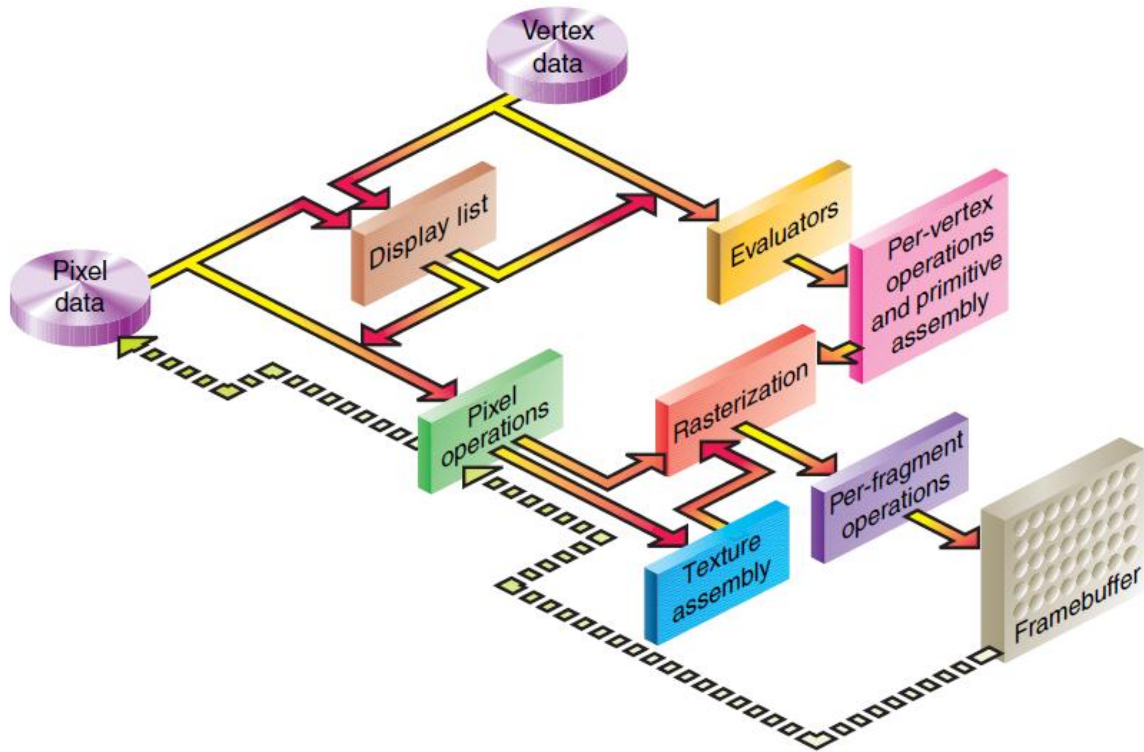

Figure 2: Oengl rendering pipeline

This is a very brief tutorial to get you started with writing 3D graphics programs using OpenGL in C. It is written to be accessible for people with a background in Java, so section 1 is a short introduction to C programming. If you have no previous knowledge of C, you will not be a C programmer after reading this, but you will be able to find your way around in existing programs, understand and edit other people's code and write small snippets of code yourself. Section 2 then shifts the focus to OpenGL, starting with its design and concepts and a small subset of its basic functionality. After the general introduction to OpenGL, we finish off with a practical tutorial, written in a very hands-on fashion. Section 3 covers the installation of all the software you need to compile and run the tutorial yourself on a Windows computer, and in section 4 we create a small OpenGL example program from the ground up.

1. A C primer for Java programmers, C is the ancestor of several modern programming languages, including C++ and Java. It is not a particularly pretty language, in fact it

allows and even encourages programmers to write very ugly programs, but it was designed to be efficient and easy to use. C is still highly useful and popular for many applications, particularly for hardware related, system-level programming. Learning the basics of C is very helpful also for understanding C++. People switching from Java to C++ are often misled by the similarities between the two languages and forget the important differences. It is of great help to a C++ programmer to know some of the pitfalls of C and to keep in mind that C++ is somewhat of an object-oriented facade built on top of C, not an object-oriented language designed from the ground up like Java. Compared to Java, C is a relatively inconvenient language to use, but it has one definite advantage: speed. Programs written in C can execute ten times faster or more than exactly the same program written in Java. The reason for that is the large overhead in Java bytecode interpretation, object manipulation and runtime safety checks. C has no such overhead, but also offers much less support for the programmer.

A familiar syntax

C has a syntax that should be immediately familiar to people who know Java. Identifiers are case sensitive, a semicolon terminates a statement, curly braces are used to group statements together, and most control constructs like if-statements, while and for loops and switch clauses look and work the same in C as in Java. Primitive types are also mostly similar: int, double, float and the like. Most operators have an identical syntax, an array index is placed within square brackets, and comments are written in the same manner. At the source code level, C and Java look very much alike, and many C statements are valid Java statements as well. Some small and strange quirks in Java like the increment operator (i=i+1 may be written i++) and the rather bizarre syntax of a for loop are directly inherited from C.

Functions and nothing else

So, C and Java may look similar, but they work differently. First of all, everything that Java has in terms of object orientation is entirely lacking in C. There are no classes, no objects, no constructors, no methods, no method overloading and no packages. In C, you have variables and functions, and that's it. You may think of it as if you only had one single class in a Java program. Furthermore, a C function must have a globally unique name. In Java, methods belong to classes and only need to have a unique name within the class, and methods within a class may be overloaded and share a common descriptive name if only their lists of parameters differ. C makes no such distinctions. One result of this is that functions with similar functionality that operate on different kinds of data must be distinguished by their names. For a large API like OpenGL, this is definitely a drawback. A vertex coordinate may be specified with two, three or four numbers, which may be of type double, float, int or short. This results in the following twelve function names being used for that single purpose:

void glVertex2d(double x, double y)

void glVertex2f(float x, float y)

void glVertex2i(int x, int y)

void glVertex2s(short x, short y)

```
void glVertex3d(double x, double y, double z)

void glVertex3f(float x, float y, float z)

void glVertex3i(int x, int y, int z)

void glVertex3s(short x, short y, short z)

void glVertex4d(double x, double y, double z, double w)

void glVertex4f(float x, float y, float z, float w)

void glVertex4i(int x, int y, int z, int w)

void glVertex4s(short x, short y, short z, short w)
```

The prefix ”gl” in the name of every OpenGL function is due to the lack of classes, separate namespaces orpackages in C. The suffixes are required to distinguish the variants from each other. In Java, these functions could all have been methods of a GL object, and all could have been given the same name: GL.Vertex(). (In fact, the Java interface to OpenGL, ”JOGL” works that way.)

Memory management

Java has a very convenient memory management system that takes care of allocating and de allocating memory without the programmer having to bother about it. C, on the contrary, requires all memory allocation and de-allocation to be performed explicitly. This is a big nuisance to programmers, and a very common source of error in a C program. However, the memory management in Java is not only a blessing. The lack of automatic memory management and other convenient functionality is one of the reasons why C and C++ are much faster than Java.

Pointers and arrays

Java by design shields the programmer from performing low level, hardware-related operations directly.

Most notably, direct access to memory is disallowed in Java, and objects are referenced in a manner that makes no assumption regarding where or how they are stored in the computer memory. C, being an older, more hardware oriented and immensely less secure language, exposes direct memory access to the programmer. Any variable may be referenced by its address in memory by means of *pointers*. A pointer is literally a memory address, denoting where the variable is stored in the memory of the computer. Pointers pointing to the wrong place in memory are by far the most common causes of error in C programs, but no useful programs can be written without using pointers in one way or another. Pointers are a necessary evil in C, and

unfortunately they are just as essential in C++. *Arrays* in Java are abstract collections of objects, and when you index an array in Java, the index is checked against upper and lower bounds to see whether the referenced index exists within the array. This is convenient and safe, but slow. C implements arrays as pointers, with array elements stored in adjacent memory addresses. This makes indexing very easy, because if you have an array arr with a number of elements and want to access arr[4], all you have to do is to add 4 times the size of one array element to the address of the first element and look at that new address. Unfortunately, no bounds checking is performed, so an index beyond the last element of an array, or even a negative index, is perfectly allowed in C and will often execute without error. Indexing out of bounds of an array could cause fatal errors later in the program, and the end result will most probably be wrong. Indexing arrays out of bounds is a very common pointer error in C, and forgetting to check for incorrect bounds in critical code is the most common security hole in C programs. Once again, C++ does not make things any better in this respect.

Source files

The source code for a C program may be split into several files, but it is not a simple task to decide what goes where, or how many files to use. In Java, the task is extremely straightforward: use one file for each class. In C, there are no classes, so you could theoretically write even a very large C program in one single file. This is inconvenient and not recommended for anything beyond the size of the small programs in this tutorial, but different programmers split their code somewhat differently into several files. There are guidelines on how to organise C source code to make it easy to understand, but no there is no single set of guidelines that everyone uses. To compensate for the lack of packages in C, there is a concept called *header files* or *include files*, which is a sort of half-baked solution that works OK most of the time. Include files are inserted by the compiler into the source code at compile time, and included header files with function prototypes can be used to provide access to extra libraries and functions that are defined in other, separately compiled source files. Unfortunately, include files can be used inappropriately, as there is nothing in the C language itself that actually mandates what should go where. Self-taught and undisciplined C programmers sometimes re-invent the wheel badly or learn from bad but ”clever” examples, and develop a programming style of their own that can be very hard to read.

main( )

A stand-alone C program must have a function called main. It is defined like this:
int main(int argc, char *argv[])

When a C program starts, its main function is invoked. Where to go from there is entirely up to the programmer. The program exits when the main function exits, or when the statement exit is executed.

Example

A very short but complete C program is given below. It calculates a function value for ten numbers from 0.0 to 0.9, stores them in an array and exits. It is of no particular use, but it shows the syntax for some common tasks.

```
/* A program that does nothing much useful */
double myfunc(double x) {
double f;
f = 3*x*x - 2*x + 8;
return f;
} // End of myfunc()
int main(int argc, char *argv[]) {int i;
double x;
double values[10];
for (i=0; i<10; i++) {
x = 0.1*i;
values[i] = myfunction(x);
} // End of for loop
return 0;
} // End of main()
```

```
/* A program that prints a countdown and exits */
#include <stdio.h>
int main(int argc, char *argv[]) {
int i;
for (i=10; i>0; i++) {
printf("%d, ", i);
}
printf("Launch!\n");
}
```

2. OpenGL: design and concept

OpenGL is a *direct rendering* API, which means that everything is drawn immediately after it is specified.

OpenGL has no memory of what has already been drawn, apart from what is painted in the rendering window. The opposite to this behaviour is *indirect rendering* or retained rendering, where the entire scene is first specified as some kind of data structure, often a tree, and only then sent away to be drawn. In OpenGL, there is no memory of the last frame, so animation has to be performed by erasing the screen and redrawing all objects at new positions. Timing of the animation is entirely up to the programmer. This makes anOpenGL program full of small, low level details, and it is necessary to move these details to separate functions for the program to be well structured and readable. Everything within OpenGL is performed by a large collection of functions starting with "gl". The header that defines most of these functions is named<GL/gl.h> on most platforms.

Primitives

There are very few graphics primitives in OpenGL. Strictly speaking, there are only four: pixel images, points, lines and triangles. Triangles are the most general and most often used 3D primitive. Several triangles can be combined to construct any planar polygon, and any surface

can be approximated by a mesh of triangles. A triangle is specified by three vertices in a 3D space. A vertex is specified by one of the many functions whose names start with ”glVertex”.

Rendering context

Apart from vertex coordinates, OpenGL has functions for specifying normal vectors, colors, texture coordinates, texture images, lights and a few other pieces of information regarding the things to be drawn.

Everything about the drawing is saved in a ”graphics state”, often called *rendering context*, that keeps track of the current color, the current normal vector, the currently active texture image and so forth. Without going into any details, we should mention that textures and lights play a very prominent part in OpenGL. Textures in particular have become very important as a means for compensating for the lack of geometric detail and the simplistic local lighting model which is still used in most real-time 3D rendering.

Transformations

Primitives alone do not make a good graphics API. Transformations are also required to make it general enough to be really useful. Transformations in OpenGL, as in most 3D graphics APIs, are described by 4x4 matrices operating on homogeneous coordinates. There are several transformation matrices in OpenGL, the most used ones being the *modelview matrix* and the *projection matrix*. The modelview matrix takes care of all the transformations from local (user) space to camera (view) space, and any transformations concerning camera motion or scene motion should be performed by changing the modelview transformation matrix. The projection matrix is responsible for the transformation from view space to screen space, most importantly the perspective projection and the transformation from general view coordinates to a pixel-oriented window coordinate system suitable for rendering. Hierarchical transformations play an important part in most 3D graphics applications. OpenGL supports hierarchical transformations by a *matrix stack*, where matrices may be pushed and popped to perform and undo transformations as required. Keep in mind that OpenGL is a direct rendering API, so only one transformation is active at a time. All that is needed is the ability to rapidly switch back and forth between different transformations, and the matrix stack makes this simple.

That’s it!

Primitives, the existence of a rendering context and knowledge of transformation matrices are the key components to understanding the underlying concept of OpenGL. It takes a lot more details to explain it in depth, but at an overview level, this is all there is to it. Now we are ready for some real programming.

3. Tools for OpenGL programming

Before you can start coding, there are a few things you need, but don’t be alarmed. The hardware you need is available in almost any modern PC, and the software can be downloaded free of charge and is easy to install.

The hardware

You should try to find a computer with a reasonably good and modern 3D graphics card to run this tutorial. Any Windows system supports OpenGL, but some less capable graphics cards will force the system to do everything in software. OpenGL is designed for hardware acceleration, so software emulation will be slow, and sometimes ugly too. Most stationary PCs sold in recent years have fairly good 3D graphics hardware, but beware of some of the cheapest graphics chips that are integrated directly onto the motherboard. Many laptops, even very modern and otherwise good systems, can also be very bad when it comes to 3D graphics.
On the other end of the scale, stationary computers designed for gaming often have outstanding 3D performance, often quite good even when compared to professional level 3D workstations.

The software

To write programs in C, you need at least a simple text editor to write the source code, and a compiler to create the executable files. A number of options exist, some of which are better than others. The tools we will use here are very good, widely used and absolutely free. The compiler is ”gcc” from the GNU project, and the editor is a full-fledged development environment for the Windows platform called “Dev-C++” from Bloodshed Software. Both are available together in a single installation package from various places over the Internet. We have placed one version where you got this document from: http://www.itn.liu.se/~stegu/OpenGLquickstart/
The file there was the most recent version from the developers as of on March 3, 2009. If you want to look for a later version, you can have a look at http://www.bloodshed.net/devcpp.html. Download and install the compiler tools by using the all-in-one installation package.

The libraries

The essential libraries you need to write OpenGL programs are already included with the Dev-C++ installation, but we will install an additional free library that makes OpenGL programming under Windows a lot easier: GLFW by Marcus Geelnardt and Camilla Berglund. You can download the source code for GLFW and compile it yourself with the gcc compiler, but to make installation easier, you can use a Dev-C++ ”development package” that will install the few files required for GLFW in the correct locations. Download the file glfw2.6-3.devPak, start Dev-C++, select Tools->Package manager, click the Install icon, locate the file glfw2.6-3.devPak and select Open, click Install, Next and Finish to perform the installation. You are now prepared to start programming with GLFW. Finally, there are a few C example files and two texture images for this tutorial, named planets1.c to planets7.c, earth.tga and random.tga. Download and save those files somewhere convenient. Now you’re all set!

The project

Start Dev-C++. Before you start coding, you need to create a new project and set it up for our particular purpose of programming in C under Windows using OpenGL and GLFW. Please follow the instructions below to do this. All the steps are required
File->New->Project... click ”Empty project”, select ”C project”, name the project ”planets”, click ”OK” and name the project file ”planets.dev”. Save the project file in a fresh folder somewhere convenient.

File->New->Source File, click ”Yes” to add the file to the project.
File->Save, name the file ”planets.c” and save it in the same folder as the project file.
Project->Project Options..., select tab “Parameters”, in text field “linker”, type
”-mwindows -lglfw -lopengl32 -lglu32” and click OK.

Our first OpenGL program

Your new and empty C source file ”planets.c” is where the C code should go. Open the file ”planets1.c” from Dev-C++, cut and paste the entire contents (see below) into your file ”planets.c” and save the file.
(Alternatively, you may include the file ”planets1.c” in the project instead of your own file, but you will want to make edits later, so it is probably best to do all your editing in ”planets.c”.)

```
#include <GL/glfw.h> // GLFW used for convenience
int main(int argc, char *argv[])
{
int running = GL_TRUE;
double t, t0, fps;
// Initialise GLFW
glfwInit();
// Open the OpenGL window
if( !glfwOpenWindow(640, 480, 8,8,8,8, 32,0, GLFW_WINDOW) )
{
glfwTerminate(); // glfwOpenWindow failed, quit the program.
return 1;
}
// Main loop
while(running)
{
// Swap buffers, i.e. display the image and prepare for next frame.
glfwSwapBuffers();
// Check if the ESC key was pressed or the window was closed.
if(glfwGetKey(GLFW_KEY_ESC) || !glfwGetWindowParam(GLFW_OPENED))
running = GL_FALSE;
}
// Close the OpenGL window, terminate GLFW and exit.
GlfwTerminate();
return 0;
}
```

Execute->Compile (or ctrl-F9, or use the “compile” button in the toolbar) to compile the program. If any errors occur, you did something wrong, probably in specifying the project options. Find the error and try again until the compilation succeeds.
Execute->Execute (or ctrl-F10, or use the “execute” button in the toolbar) to execute your newly compiled file. Because what you create is a standard Windows executable file, you can also

execute the program without any special tools at all. Open a file explorer window, locate the program file ”planets.exe” in your project folder and double-click on it.
If an empty black window is showing, everything works OK and you’re ready to continue! Close your newly created application by pressing the ESC key or by closing its window.
Make a habit of always closing your applications when you are done testing. A typical OpenGL program uses a lot of processing and graphics power, and having more than one such application running, either as a visible window or minimized in the task bar, will slow down your other programs a lot.

CODE IMPLEMENTED FOR PRACTISE

OUTPUT:

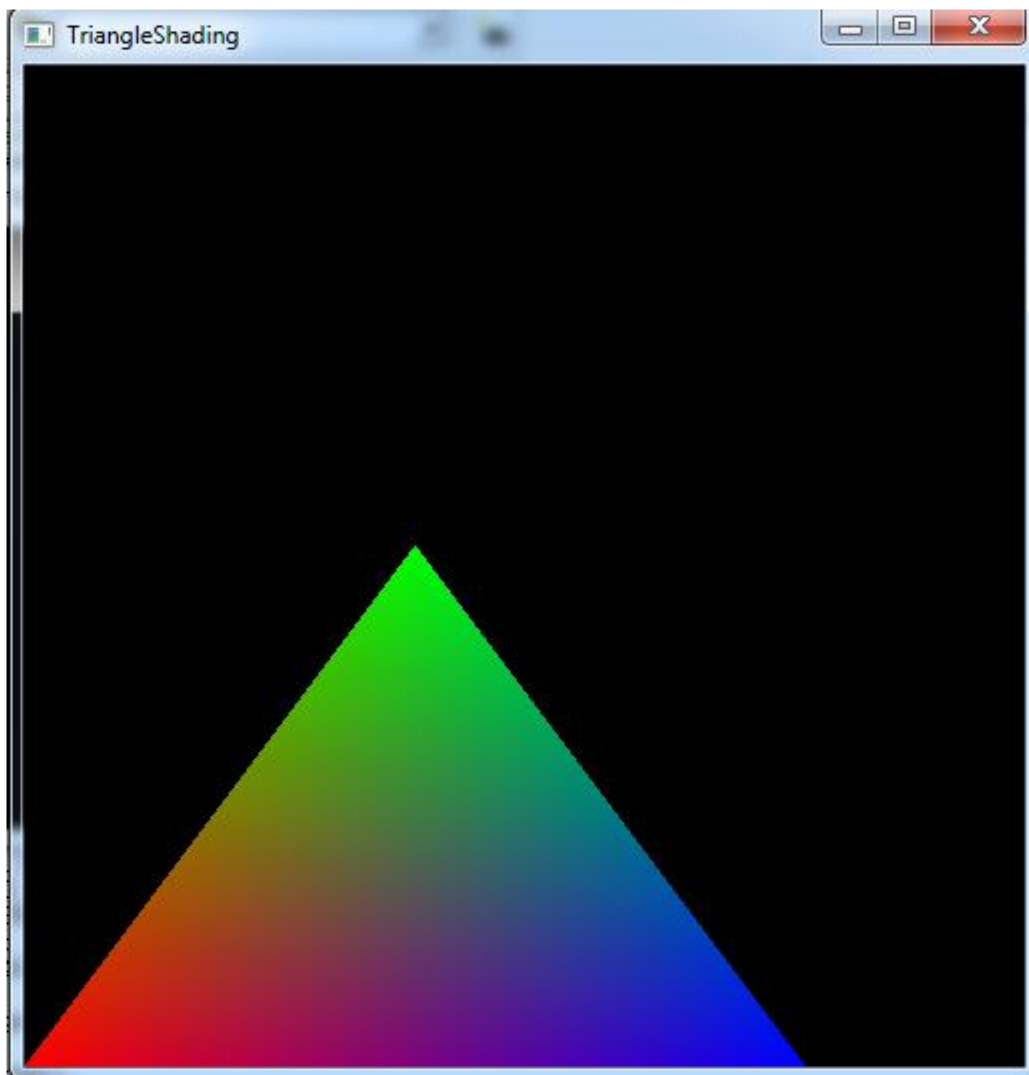

Figure 3: Learning to shade in Opengl

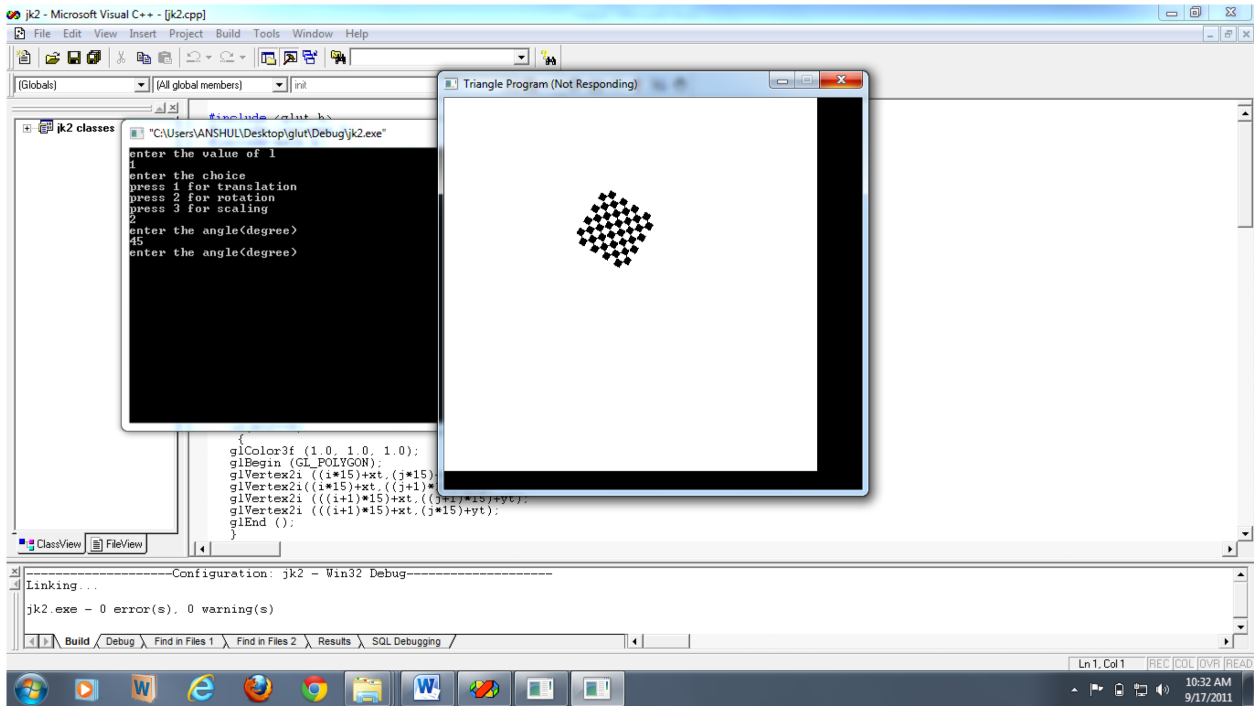

Figure 4: Implementing Scaling

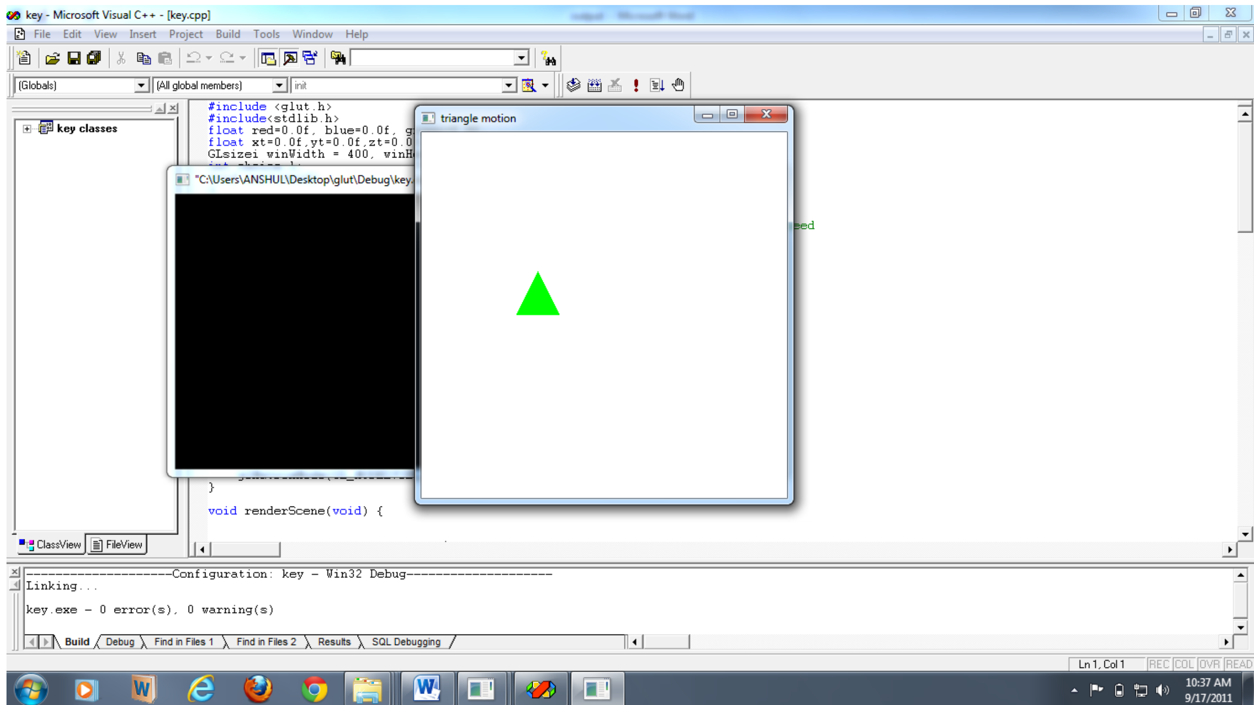

Figure 5: Implementing key movement and color change and zooming

A less boring OpenGL program

Our first OpenGL program is really boring, because it doesn't draw anything. Let's add some graphics to the window. We need to set up a camera projection, and draw at least one primitive. This can be accomplished by the code in "planets2.c". Cut and paste, compile and have a look at the result. Now take a closer look at the code. The drawing code that was added compared to "planets1.c" is in main and consists of two parts: first the window is prepared for drawing by erasing it and setting up a camera projection, then a single triangle is drawn by a sequence of vertices enclosed between a glBegin and a glEnd. Before each vertex is specified, the graphics context is changed by glColor to specify a color. When a color is specified, subsequent vertices will have that color if no new color is specified. (If no color is ever specified, OpenGL starts with white as the current color.) Note that the OpenGL rendering performs smooth color interpolation over the surface to set the color between vertices.

An FPS counter

The image in the example above does not change over time, but it is actually redrawn many times per second. To find out exactly how many times per second it is updated, we can add a frame counter to the program and use the built-in timer function in GLFW to calculate the number of frames per second (FPS).

We have no convenient text output in our application, so we print the FPS counter information in the windowtitle bar. This is performed by a couple of extra variables and the function showFPS in "planets3.c" Try it out. The display is exactly the same as before, but now the program keeps track of how fast it is updated. The FPS reading is probably some number reasonably close to 60, 75 or 85 FPS, because OpenGL by default updates the graphics at most once per display refresh. If you want to see how fast your simple scene can be drawn, uncomment the following function call immediately before the main loop:

```
glfwSwapInterval(0);
```

This will make OpenGL redraw the display as fast as possible, without waiting for a new display scan. You will probably see a great increase in FPS reading when you do this change. The scene we are drawing so far is extremely simple and can be updated thousands of times per second on a good computer. If you make the window smaller, the FPS reading will increase, and if you make it larger, the FPS will decrease. This is because OpenGL primitives take longer to render the more pixels they cover, and erasing a large window also takes longer than a small window. One of the many limiting factors to OpenGL hardware rendering performance is how fast pixels can be written to the frame buffer memory.

An animated object

We are constantly redrawing the image, so it makes sense to animate the scene to make use of all that rendering power. All you need to do is make some aspects of the rendering dependent on time. We can use the GLFW timer to keep track of time, and rotate the object by a transformation. Uncomment the following line before the call to glBegin and you should see a rotating triangle instead.

```
glRotatef(90.0f*glfwGetTime(), 0.0f, 0.0f, 1.0f);
```

The rotation speed is 90 degrees per second, and the rotation is performed around the axis (0,0,1), i.e. The Z

axis. Note that, contrary to many other programming APIs, OpenGL has chosen to measure angles in degrees rather than radians.

A more complicated object

Our program has grown since we started, and it is time to move some stuff out of the main function to make it more readable. It is also time to draw something in true 3D - the triangle we have drawn up until now is only a flat object in the (x,y) plane. The file ”planets4.c” contains a more complicated object: a colorful cube with six faces. Try it out and browse through the code.

Hierarchical transformations

The cube in ”planets4.c” rotates around the Z axis, and the camera is looking along the Y axis. To make the view a little more interesting, you can add a static rotation immediately before the animated rotation, by simply adding the following line before the existing call to glRotatef:

```
glRotatef(30.0f, 1.0f, 0.0f, 0.0f);
```

Hierarchical transformations are a key to efficient 3D graphics programming. Let's try that in OpenGL. To make a planetary system with a sun and a planet, we could replace the drawing commands with this:

```
float t = (float)glfwGetTime();
glRotatef(30.0f, 1.0f, 0.0f, 0.0f);
glPushMatrix();
glRotatef(90.0f*t, 0.0f, 1.0f, 0.0f);
drawColorCube(1.0);
glPopMatrix();
glRotatef(30.0f*t, 0.0f, 1.0f, 0.0f);
glTranslatef(3.0f, 0.0f, 0.0f);
glRotatef(180.0f*t, 0.0f, 1.0f, 0.0f);
drawColorCube(0.5);
```

The order in which the transformations are applied matters. A rotation followed by a translation will translate the object along the new, transformed coordinate axes. A translation followed by a rotation will rotate the object at a new position, but keep the rotation axis in the same place relative to the object. Now try to extend the small planetary system with acubical moon orbiting around the planet. You may have to adjust the size, position and rotation speed of the objects to make the scene look good.

Draw a sphere

Real celestial bodies like stars and planets are not cubical, but approximately spherical. Drawing a sphere in OpenGL means approximating it with polygons. It is not difficult to do this, but it takes quite a bit of work to get it right. Instead of asking you to do the sphere drawing function yourself, we present you with a finished function with the name drawColorSphere to illustrate the principle. You can find it in the file “planets5.c”. Compile and run the file, and have a good look at the new code in the function drawColorSphere.

Like drawColorCube, the size of the sphere is one parameter to the function, and an additional integer parameter makes it possible to choose the detail level of the polygonal approximation.

Try varying the detail level from around 2 to around 20 to see what happens with the shape of the “sphere” object. If you ask for a detail level of 100, how many polygons will be drawn for each sphere? How high can you go in the number of polygons before your computer starts to drop the frame rate below an acceptable speed and make the animation jerky? (Note that the main workload of the drawing is not the graphics, but rather the large amount of calls to sin() and cos() functions for each frame. There are smarter ways of drawing a sphere in OpenGL.)
Use the sphere to create your own planetary system with a sun, a planet and a moon. Spend more triangles for large spheres, and fewer for small ones to make the display look nice without using an unnecessarily high number of polygons.

A texture

Without textures for the object, the scene looks rather dull. In OpenGL, it is easy to add a texture to an object. All it takes its that you enable texturing, load a texture into memory, activate a texture and make sure to supply texture coordinates with your vertices. The hardest part to implement yourself is the reading of pixel data from an image file of some sort.Once that is taken care of, activating and using a texture is easy. Texture images stored in TGA bitmap files can be read in by GLFW, so we will use the convenience function glfwLoadTexture2D to load and activate a texture. (Without the aid of GLFW, loading and activating a texture will require a few more lines of code, but will also be more flexible.) Code to activate and use a texture can be found in the file planets6.c. Use that code as a starting point for doing your own planetary system as defined below.

Optional assignment: Light sources

Lighting is an important part of 3D graphics, and OpenGL has built-in support for three types of lights: directional lights, point lights and spotlights. Until now, we have kept all lighting disabled, and then objects are simply rendered in the current vertex color, as if there was a full strength white ambient light. When lighting is enabled, vertex normals and the positions and properties of currently active light sources also affect the color. If texturing is also enabled, the texture color is applied to the object first, and then multiplied by the result from the lighting calculations.
Code to activate and place a point light source can be found in planets7.c. Modify it to suit your needs and make your solar system appear as if it were all lit by the sun in the middle:
Activate and place a point light source in the middle of the sun, set a white base color for all objects, use textures for the sun, the planet and the moon. Draw the sun with lighting disabled (which will yield full intensity), but draw the planet and the moon with lighting enabled. Hint: you can use some ambient light to make the shadow side of the planet and the moon appears somewhat lighter than pitch black. Ambient light can be set by the following statements:

```
float color4f[4]={0.1f, 0.1f, 0.1f, 1.0f};
glLightfv(GL_LIGHT0, GL_AMBIENT, color4f);
```

More information

If you want to go further and explore more of the endless possibilites with OpenGL, there is a lot of information on the Internet. One very popular set of tutorials is NeHe. They are easy to follow

and thorough, but they are based on an older and outdated OpenGL framework called GLUT, so you might want to adjust them somewhat to use GLFW instead. If you are serious about learning more about OpenGL, a highly recommended source of knowledge is “OpenGL Programming Guide”, published by Addison-Wesley. There is also the “OpenGL Reference Manual” from the same publisher, if you want a full alphabetical list of all OpenGL functions. Before you buy either of those books, however, have a look at http://www.opengl.org, where you can find older versions in HTML and PDF format. Contrary to Direct3D, OpenGL has not undergone any dramatic architectural changes over the years, so older versions of OpenGL documentation are still perfectly useful for learning the basics. Everything written about OpenGL since version 1.1 from the early 1990's is still a source to learn from. Have fun!

Analysis and Design

4.1 GUI

Opengl being cross platform is available for all kind of users. With the improvement in the Graphical User Interface of the Unix/Linux environment we were freely able to use the GUI for our project. The user of the software game or the players will be able to move their pawns first by selecting the desired pawn to be moved which will change the color of the block and will also highlight the block then on selecting the block in which the pawn is to be placed next.

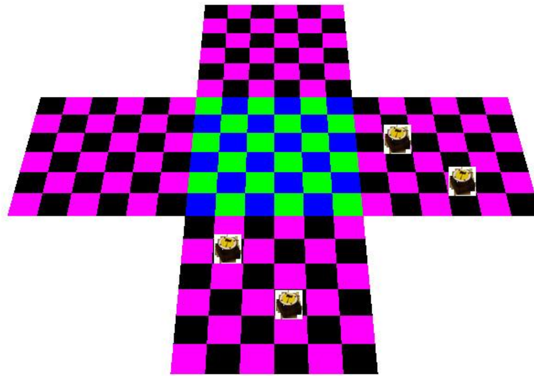

Figure: 3 Game Board

The dead pawn or the pawn being eliminated will be placed to the side of the player playing the game.

4.2 Input and Controls

As already mentioned the game is only controllable by the mouse by selecting the pawn to be moved and then reselecting the location where the pawn is to be moved.

4.3 View of the Game Board

The player can select any of the four sides by choosing an option in the menu that corresponds to the four different colored pawns.

A perspective projection of the game board has been shown; however during the earlier stages of development only orthographic projection of the game board was shown.

4.4 Texture

The player can also exercise an option given in the menu for changing the look of the board and the pawns. Three different textures have been provided – porcelain, marble and wood for the game board and porcelain, frosted glass and wood for the pawns.

The textures have been downloaded from the internet and many more of them are available which has been imported using an opengl program.

4.5 Pawns

There were two possible options available for the pawns either to design it from the scratch using basic opengl functions or to import the image of the pawns on cubes and then treat them as pawns. Both of them having there complexities and appearance advantages. However we selected importing pawns as the better option depending upon our limitations as designing the pawns from the scratch would have been too time consuming and adding little to our programming skills.

4.6 Treasure keys

Similar approach to that of pawns was adopted for the treasure keys as well.

4.7 Treasure maps

Treasure maps also called as cue cards have been designed using simple square generating functions and then writing text using straight lines in these cards.

4.8 Dice

The function of the dice was acquired using a simple random number generator in opengl. However to make it look more better a 3D cube was designed and then numbers shown on them using simple points and controlling their size. The user clicks on the dice to start moving it which rotates in the random direction and stops when user clicks on it again thereby selecting the number which comes on the face of the dice and then moving the pawns it desires depending on the game play.

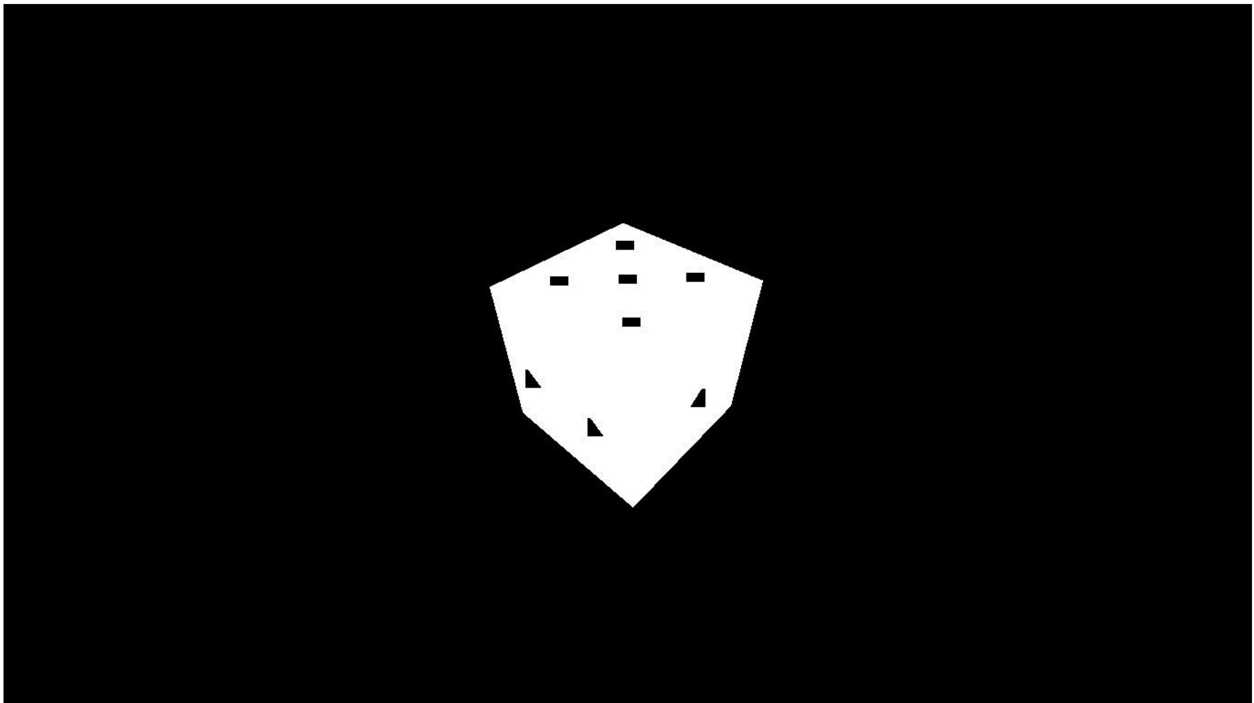

Figure: 4 Animated Dice

Development Stages

There were several stages of development during the entire duration of project design and implementation. Initially we acquainted ourselves with the basic knowledge of the working of the Opengl API. Here we tried to implement several codes that clarifies any doubts and also explain the functionalities and intricacies of Opengl.

Few of those were:

How to draw a simple square.

How to make simple and then complex figures using the basic opengl function for drawing points, lines and polygons.

How to draw a cube.

How to selects a pixel using mouse and then the similar concept was translated further in developing a module which helps us select a block to move the pawn to the desired location.

How to draw a simple cube.

How to draw a cube with textured surface.

How to make the cube rotate in arbitrary direction for creating our dice.

How to make pawns using file import program and how to draw any object by using points, lines and polygons.

How to control the viewing position and the light source, blending, anti-aliasing, creating shadows, double buffering (which has not been implemented we are looking forward to implement it as we want the pawns to have an animated motion).

Module Specification

6.1 Game board

This module implements the game board. It has several functions for implementing the texture. Color_Texture, Material_Texture, Border_Texture, Do_Border, Do_Board, Do_Rotate, Do_Piesce, Do_Display.

The functions Color_Texture, Material_Texture and Border_Texture implements the texture in the game board after the board has been drawn using a vertex array 'pattern' and polygon stipple pattern is generated.

GLfloat Blackamb[4], GLfloat blackdif[4], GLfloat blackspec[4] whiteamb[4], GLfloat whitedif[4], GLfloat whitespec[4], GLfloat copperamb[4], GLfloat copperdif[4], GLfloat copperspec[4], GLfloat darkamb[4] GLfloat darkdif[4], GLfloat darkspec[4] stores the location of the type of light on the board for generating a textured effect.

The Do_Boarder function designs the border in the game board. Similarly the Do_Board function implements the board and it also calls all other funtions to do the actual drawing.

Do_Rotate function helps in setting the required specification for the orientation of the game board.

Do_Display finally renders the game board which further calls all other functions.

Other predefined functions include –

glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);

glBindTexture(GL_TEXTURE_2D, texName[2]);

glCallList(texName[1]);

6.2 Pawns

This module loads the desired image on a cube which serves as a pawn. The images have been downloaded from the internet. The function loadBMP (const char* filename) loads the file.

An array of the form (R1, G1, B1, R2, G2, B2 ...) indicates the color of each pixel in image. Color components range from 0 to 255. The array starts the bottom-left pixel, and then moves

right of the end of the row, then moves up to the next column, and so on. This is the format in which OpenGL likes images.

6.3 Dice

This module uses basic opengl functions like glrotatef() to rotate the cube where one of the parameters is modified continuously after each draw call and the image is redrawn. The cube is drawn using detailed functions rather than calling the inbuilt Cube function.

6.4 Treasure Keys

Steps similar to that of pawns designing and implementation have been taken by loading an image and treating it as our object.

6.5 Treasure Maps

Treasure maps are incorporated by loading an image and then writing the text meant for the map.

Testing, Debugging and Errors

The game is encountered with different possible errors where proper solutions are still missing. Unexpected pawn moves during the game play, unable to identify the winner, and loss of priorities of the pawns by the implementation code are some of the existing errors encountered. However the basic functionality of the game is still visible and we can see moving boards and pieces and accuracy of game logic to a certain level.

Only experience based testing is implemented due to lack of time and no regression based testing is done as the concept is out of picture given the functionality have yet not been totally implemented.

Future Development and Extensions

The game shows us a lot of promise not only in terms of adding extra functionalities but also launching it as a full-fledged game.

We can not only make an improvement on the level of graphics being implemented, but also improve the game in lot of other areas such as designing the pawns using opengl instead of importing the image which will make our game look more real life like, we can introduce animation while moving the pawns when it is relocated from one place to another.

Another very interesting feature that can be included in possibly all the software games is artificial intelligence, where the game will be saving the data of all best possible moves generated during the game play and use it against the player in one of the gaming modes when the player is playing against the computer.

Different levels can be provided where the player can have the liberty to select out of various levels being given as an option depending upon its skills in the game.

Voice incorporation in the game indicating various events occurring during the game play to make it more realistic.

Limitations

The lack of time is the single biggest challenge hindered the completion of the game though we are looking forward to complete the game even after our graduation due to our growing interest in the subject and its advantages in our career growth.

Unavailability of the feature in opengl by which we could lift an object and move around the drawing screen also was a major problem faced by us and its solution is still missing, fortunately for us it exists and soon we will be able to find our way around it if the exact solution is not known

Division of Work

Both of the team members have given equal responsibilities leading to the completion of the project. Jitendra Kalwaniyan had a major role in design of the game board and the Dice. Anshul Gupta covered addition of texture to the game, moving the pieces using mouse and importing the image for developing the pawns.

Both of us shared the responsibilities equally and almost all the work was done together helping each other out in all codes without exception.

Conclusion

During the development of our project we learned two most essential things how to go about a software development and secondly the importance of team work.

We gained a good amount of experience in opengl and its importance in graphics technology and game development. Its ease of use in designing complex shapes in easy steps and also to its fast draw calls which makes the game more faster without any lag.

Appendix A

Code:

Simple game board -

```
#include<iostream.h>
#include<glut.h>
#include <windows.h>
#include <stdio.h>
#include <gl.h>
#include <glu.h>
#include <glaux.h>

void init(void)
{
        glClearColor(0.1,0.1,0.1,0.1);
        glMatrixMode(GL_PROJECTION);
        gluOrtho2D(0.0,200.0,0.0,150.0);
}


void Graphics (void)
{
        glClear (GL_COLOR_BUFFER_BIT);
        GLubyte pattern[]={
        0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55,
   0xAA, 0xAA, 0xAA, 0xAA, 0x55, 0x55, 0x55, 0x55
        };
        glPolygonStipple(pattern);
        glEnable(GL_POLYGON_STIPPLE);
```

```
        glBegin(GL_POLYGON);
        glColor3f (0.7, 0.7, 0.7);
        glVertex2f(20, 20);
        glVertex2f(180, 20);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(180, 130);
        glVertex2f(20, 130);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(40, 20);
        glVertex2f(60, 20);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(60, 40);
        glVertex2f(40, 40);
        glEnd();


        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(80, 20);
        glVertex2f(100, 20);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(100, 40);
        glVertex2f(80, 40);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(120, 20);
        glVertex2f(140, 20);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(140, 40);
        glVertex2f(120, 40);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(160, 20);
        glVertex2f(180, 20);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(180, 40);
        glVertex2f(160, 40);
        glEnd();
//////////////////////////////////////////
```

```
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(20, 40);
        glVertex2f(40, 40);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(40, 60);
        glVertex2f(20, 60);
        glEnd();


        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(60, 40);
        glVertex2f(80, 40);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(80, 60);
        glVertex2f(60, 60);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(100, 40);
        glVertex2f(120, 40);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(120, 60);
        glVertex2f(100, 60);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(140, 40);
        glVertex2f(160, 40);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(160, 60);
        glVertex2f(140, 60);
        glEnd();
////////////////////////////////////
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(40, 60);
        glVertex2f(60, 60);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(60, 80);
        glVertex2f(40, 80);
        glEnd();
```

```
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(80, 60);
        glVertex2f(100, 60);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(100, 80);
        glVertex2f(80, 80);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(120, 60);
        glVertex2f(140, 60);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(140, 80);
        glVertex2f(120, 80);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(160, 60);
        glVertex2f(180, 60);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(180, 80);
        glVertex2f(160, 80);
        glEnd();
//////////////////////////////////
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(20, 80);
        glVertex2f(40, 80);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(40, 100);
        glVertex2f(20, 100);
        glEnd();


        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(60, 80);
        glVertex2f(80, 80);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(80, 100);
        glVertex2f(60, 100);
        glEnd();
```

```
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(100, 80);
        glVertex2f(120, 80);
   glColor3f (0.1, 0.1, 0.1);
        glVertex2f(120, 100);
        glVertex2f(100, 100);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(140, 80);
        glVertex2f(160, 80);
   glColor3f (0.1, 0.1, 0.1);
        glVertex2f(160, 100);
        glVertex2f(140, 100);
        glEnd();
////////////////////////////////////////
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(40, 100);
        glVertex2f(60, 100);
   glColor3f (0.1, 0.1, 0.1);
        glVertex2f(60, 120);
        glVertex2f(40, 120);
        glEnd();


        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(80, 100);
        glVertex2f(100, 100);
   glColor3f (0.1, 0.1, 0.1);
        glVertex2f(100, 120);
        glVertex2f(80, 120);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(120, 100);
        glVertex2f(140, 100);
   glColor3f (0.1, 0.1, 0.1);
        glVertex2f(140, 120);
        glVertex2f(120, 120);
        glEnd();
```

```
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(160, 100);
        glVertex2f(180, 100);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(180, 120);
        glVertex2f(160, 120);
        glEnd();
//////////////////////////////////
        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(20, 120);
        glVertex2f(40, 120);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(40, 140);
        glVertex2f(20, 140);
        glEnd();


        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(60, 120);
        glVertex2f(80, 120);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(80, 140);
        glVertex2f(60, 140);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(100, 120);
        glVertex2f(120, 120);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(120, 140);
        glVertex2f(100, 140);
        glEnd();

        glBegin(GL_POLYGON);
        glColor3f (0.9, 0.9, 0.9);
        glVertex2f(140, 120);
        glVertex2f(160, 120);
    glColor3f (0.1, 0.1, 0.1);
        glVertex2f(160, 140);
        glVertex2f(140, 140);
        glEnd();
```

```
        glDisable(GL_POLYGON_STIPPLE);
        glFlush();
}


void main(int argc, char**argv)
{
        glutInit (&argc, argv);
        glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
        glutInitWindowPosition (50,100);
        glutInitWindowSize (800,600);
        glutCreateWindow("");
        init();
        glutDisplayFunc (Graphics);
        glutMainLoop();
}
```

Animated Dice –

```
#include <windows.h>
#include <stdio.h>
#include <glut.h>
#include<math.h>
//#include<glFont.h>
GLsizei winWidth = 400, winHeight = 400;

float xrot=0.0;
float yrot=0.0;
float zrot=0.0;
float ratio;

void init(void)
{
        glClearColor (0.0, 0.0, 0.0, 0.0);// set white background color
glMatrixMode (GL_PROJECTION);
gluOrtho2D (-400.0, winWidth, -400.0, winHeight);
glEnable(GL_DEPTH_TEST);
}

void reshape( int w, int h )
{

        if(h == 0)
                h = 1;

        ratio = 1.0f * w / h;
```

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

        // Set the viewport to be the entire window
   glViewport(0, 0, w, h);

        // Set the clipping volume
        gluPerspective(80,ratio,1,200);
        glMatrixMode(GL_MODELVIEW);
        glLoadIdentity();
        gluLookAt(0, 0, 30,
                  0,0,10,
                        0.0f,1.0f,0.0f);
}

void display( void )
{
        glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glLoadIdentity ( );
  glPushMatrix();
  glTranslatef ( 0.0, 0.0, -5.0 );
  glRotatef ( xrot, 1.0, 0.0, 0.0 );
  glRotatef ( yrot, 0.0, 1.0, 0.0 );
  glRotatef ( zrot, 0.0, 0.0, 1.0 );

  glBegin ( GL_POLYGON );
  glColor3f (1.0, 1.0, 1.0);
      //front
  glVertex3f(-1.0f,-1.0f,  1.0f);
  glVertex3f( 1.0f, -1.0f,  1.0f);
  glVertex3f( 1.0f,  1.0f,  1.0f);
  glVertex3f(-1.0f,  1.0f,  1.0f);
                // Back

  glVertex3f(-1.0f, -1.0f, -1.0f);
  glVertex3f(-1.0f,  1.0f, -1.0f);
  glVertex3f( 1.0f,  1.0f, -1.0f);
  glVertex3f( 1.0f, -1.0f, -1.0f);
                // Top Face
  glVertex3f(-1.0f,  1.0f, -1.0f);
  glVertex3f(-1.0f,  1.0f,  1.0f);
  glVertex3f( 1.0f,  1.0f,  1.0f);
  glVertex3f( 1.0f,  1.0f, -1.0f);
// Bottom Face
  glVertex3f(-1.0f, -1.0f, -1.0f);
  glVertex3f( 1.0f, -1.0f, -1.0f);
```

```
   glVertex3f( 1.0f, -1.0f,  1.0f);
   glVertex3f(-1.0f, -1.0f,  1.0f);
                   // Right face
   glVertex3f( 1.0f, -1.0f, -1.0f);
   glVertex3f( 1.0f,  1.0f, -1.0f);
   glVertex3f( 1.0f,  1.0f,  1.0f);
   glVertex3f( 1.0f, -1.0f,  1.0f);
          //left

   glVertex3f(-1.0f, -1.0f, -1.0f);
   glVertex3f(-1.0f, -1.0f,  1.0f);
   glVertex3f(-1.0f,  1.0f,  1.0f);
   glVertex3f(-1.0f,  1.0f, -1.0f);
         glEnd();


         glColor3f(0.0,0.0,0.0);
          glPointSize(20.0);
glBegin(GL_POINTS);
glVertex3f(-0.5f,1.0f,0.5f);
glVertex3f(0.5f,1.0f,0.5f);
glVertex3f(-0.5f,1.0f,0.0f);
glVertex3f(0.5f,1.0f,0.0f);
glVertex3f(-0.5f,1.0f,-0.5f);
glVertex3f(0.5f,1.0f,-0.5f);

glVertex3f(0.0f,-1.0f,0.0f);
glVertex3f(0.5f,-1.0f,-0.5f);
glVertex3f(-0.5f,-1.0f,-0.5f);
glVertex3f(0.5f,-1.0f,0.5f);
glVertex3f(-0.5f,-1.0f,0.5f);

glVertex3f(1.0f,-0.5f,0.5f);
glVertex3f(1.0f,-0.5f,-0.5f);
glVertex3f(1.0f,0.5f,0.5f);
glVertex3f(1.0f,0.5f,-0.5f);

glVertex3f(0.5f,0.0f,-1.0f);
glVertex3f(-0.5f,0.0f,-1.0f);
glVertex3f(0.0f,0.0f,-1.0f);

glVertex3f(0.5f,0.0f,1.0f);
glVertex3f(-0.5f,0.0f,1.0f);

glVertex3f(-1.0f,0.0f,0.0f);
```

```
glEnd();
  glFlush();


  glPopMatrix();
  xrot+=0.30f;
        yrot+=0.30f;
        zrot+=0.00f;
  glutSwapBuffers();
}

void keyboard ( unsigned char key, int x, int y )  // Create Keyboard Function
{
  switch ( key ) {
    case 27:        // When Escape Is Pressed...
      exit ( 0 );   // Exit The Program
      break;
        case 48:
                xrot+=1.0f;
        //yrot+=1.0f;
        //zrot+=0.0f;
                break;
        case 49:
                //xrot+=0.30f;
        yrot+=1.0f;
        //zrot+=0.0f;
        break;
        case 50:
                //xrot+=0.30f;
        //yrot+=1.0f;
        zrot+=1.0f;
        break;
                // Ready For Next Case
    default:        // Now Wrap It Up
      break;
  }
}

void arrow_keys ( int a_keys, int x, int y )  // Create Special Function (required for arrow keys)
{
  switch ( a_keys ) {
    case GLUT_KEY_UP:     // When Up Arrow Is Pressed...
      glutFullScreen ( ); // Go Into Full Screen Mode
      break;
    case GLUT_KEY_DOWN:               // When Down Arrow Is Pressed...
      glutReshapeWindow ( 500, 500 ); // Go Into A 500 By 500 Window
```

```
      break;
    default:
      break;
  }
}

void main ( int argc, char** argv )   // Create Main Function For Bringing It All Together
{
  glutInit            ( &argc, argv ); // Erm Just Write It =)
  glutInitDisplayMode ( GLUT_DEPTH | GLUT_DOUBLE | GLUT_RGBA );
  glEnable(GL_DEPTH_TEST);// Display Mode
  glutInitWindowPosition (0,0);
  glutInitWindowSize  ( 500, 500 ); // If glutFullScreen wasn't called this is the window size
  glutCreateWindow    ( "dice" ); // Window Title (argv[0] for current directory as title)
  init ();
  glutFullScreen      ( );          // Put Into Full Screen
  glutDisplayFunc     ( display );  // Matching Earlier Functions To Their Counterparts
  glutReshapeFunc     ( reshape );
  glutKeyboardFunc    ( keyboard );
  glutSpecialFunc     ( arrow_keys );
  glutIdleFunc              ( display );
  glutMainLoop        ( );          // Initialize The Main Loop
}
```

Loading an image in Opengl -

```
#ifndef IMAGE_LOADER_H_INCLUDED

#define IMAGE_LOADER_H_INCLUDED

//Represents an image

class Image
{

        public:

                Image(char* ps, int w, int h);

                ~Image();
/* An array of the form (R1, G1, B1, R2, G2, B2, ...) indicating the color of each pixel in image.
Color components range from 0 to 255. The array starts the bottom-left pixel, then moves right to
the end of the row, then moves up to the next column, and
so on.  This is the format in which OpenGL likes images. */
```

```
        char* pixels;
        int width;
        int height;

};
//Reads a bitmap image from file.
Image
loadBMP(const char* filename);
#endif
```

Appendix B

References:

1. **Opengl programming guide, Dave Shreiner, Seventh edition, opengl version 3.1 and 3.2.**

2. **https://www. Opengl.org**

3. **Computer Graphics with Opengl, Third Edition, Hearn and Baker.**

4. **Opengl Super Bible, Richard S. Wright, Jr., Michael Sweet.**

5. **https://www.glgameprogramming.com/red**