The Implementation of a 3D Snooker Table Using OpenGL

Ying Luo

A Major Report

in

Department

of

Computer Science

Presented in Partial Fulfillment of the Requirements
For the Degree of Master of Computer Science
Concordia University
Montreal, Quebec, Canada

April 2004

Your file Votre référence
ISBN: 0-612-91075-X
Our file Notre référence
ISBN: 0-612-91075-X

Abstract

The Implementation of a 3D Snooker Table Using OpenGL

Ying Luo

This project demonstrates a 3D snooker table coded by Visual C++ Version 6 and OpenGL on Windows 9x/2000/XP environment. It is concerned with constructing partial components for the snooker game rather than creating entire playable game. The purpose of the project is to get a 3D model of a snooker table and learn more about object-oriented modeling and design methodology. This report will explain how the simulation was created and modeled, and the OpenGL techniques used during the construction of the simulation. Finally, future work is presented.

Acknowledgement

I am here with thanks to my supervisor, Professor Peter Grogono, for his patience and constant guidance throughout the year and for making this project an enjoyable one.

I am grateful to the professors and staff in Computer Sciences department at Concordia for the wonderful courses and services. I would especially like to say thanks to Ms. Halina Monkiewicz, who was always friendly and prompt in providing assistance.

Table of Contents

1 INTRODUCTION 1

1.1 MOTIVATION 1

1.2 SCOPE 1

1.3 THE DESIGN METHODOLOGY 2

1.4 ORGANIZATION OF THE REPORT 2

2 PROJECT BACKGROUND 3

2.1 OVERVIEW 3

2.2 PLATFORM AND PROGRAMMING LANGUAGE 4

3 TECHNICAL BACKGROUND 8

3.1 3D GRAPHICS FUNDAMENTALS 8

3.1.1 3D perception 8

3.1.2 2D + Perspective = 3D 9

3.2 GRAPHIC COORDINATE SYSTEM 10

3.3 OPENGL OVERVIEW 12

3.3.1 What is OpenGL? 12

3.3.2 The OpenGL Model 14

3.4 GRAPHIC CONCEPTS 16

3.4.1 The Color Model 16

3.4.2 The Matrix Model 19

3.4.3 The Viewing Model 21

3.4.4 The Lighting Model and Surface Properties 23

3.4.5 Curves and Surface model 24

3.4.6 Texture mapping 27

4 PROJECT IMPLEMENTATION 30

4.1 CREATE AN OPENGL WINDOW 30

4.2 MODEL THE TABLETOP 32

4.3 CREATION OF POCKETS 35

4.4 TABLE EDGE GENERATION 37

4.5 VIEWPOINTS MOVEMENT 40

4.6 LIGHTING 48

4.7 SHADOW OF THE TABLE EDGE 49

5 CONCLUSION 52

5.1 INTRODUCTION 52

5.2 EVALUATION 52

5.3 EXPERIENCES IN OPENGL 53

5.4 SUGGESTIONS FOR FUTURE RESEARCH 54

1. INTRODUCTION

1.1 Motivation

This project demonstrates a 3D snooker table with different views. It is concerned with constructing partial components for the snooker game rather than creating an entire playable game. The purpose of the project is to obtain a 3D model of a snooker table and to learn more about object-oriental modeling and design methodology.

1.2 Scope

The project explores several areas of graphics computing including:

- Viewing and Modeling Transformation
- Projection Transformation
- Texture Mapping
- Lighting
- Shadow

Viewing transformation is analogous to positioning and aiming a camera. Modeling transformation is analogous to positioning and orienting the model, or objects to be drawn. Viewing transformations must precede modeling transformations in OpenGL code. Specifying the projection transformation is like choosing a lens for a camera. It is possible to think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. Texture mapping means the mapping of a function onto a surface in 3-D. The domain of the function can be one, two, or three-dimensional, and it can be represented by either an array or by a mathematical function. OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components that are reflected in various directions. The shadow of an object on a surface is formed by projecting the object onto the plane of the surface, using the light source as the center of projection. [1]

1.3 The design Methodology

With graphic projects, major alterations often require very few lines of new code; therefore it is often quicker to experiment with code rather than to take time to designing the solution. I called this technique 'design by coding'.

1.4 Organization of the Report

This report is divided into five chapters. Chapter one is an introduction. It briefly describes the aim and objectives of this report. Chapter two describes the background and underlying concepts of the project. Chapter three starts with an introduction to 3D graphic concepts and continues with my implementation tools — OpenGL. Chapter four describes the detailed implementation of the model; selected parts of the implementation are listed in this chapter. The last chapter concludes with an evaluation of the result and suggests further work for the project.

2 PROJECT BACKGROUND

2.1 Overview

A real snooker table consists of:

- A tabletop, which is covered by green cloth, is in a shape of rectangle. The size of snooker tables is bigger than pool tables.
- Six pockets, which are covered by green cloth, each in the shape of an irregular circle. The area surrounding of the pockets is covered by grey metal.
- Four edges of the tabletop, which are covered by green cloth, dark brown wood and grey metal, are in the shape of rectangle.
- Four table legs, which are made by wood, usually are in fancy shape and very beautiful.

There is also a floor with yellow wood texture included in the project.

On the point of scale, there is lots of difference between snooker and pool. In this project, as Dr. Grogono specified, the table has the following dimensions:

Length = 366 cm

Width = 201 cm

These are outside measurements. The playing surface (that is, the area on the top in which the balls can roll) is 350 x 175cm, as the ***Figure 2.1-1*** shows.

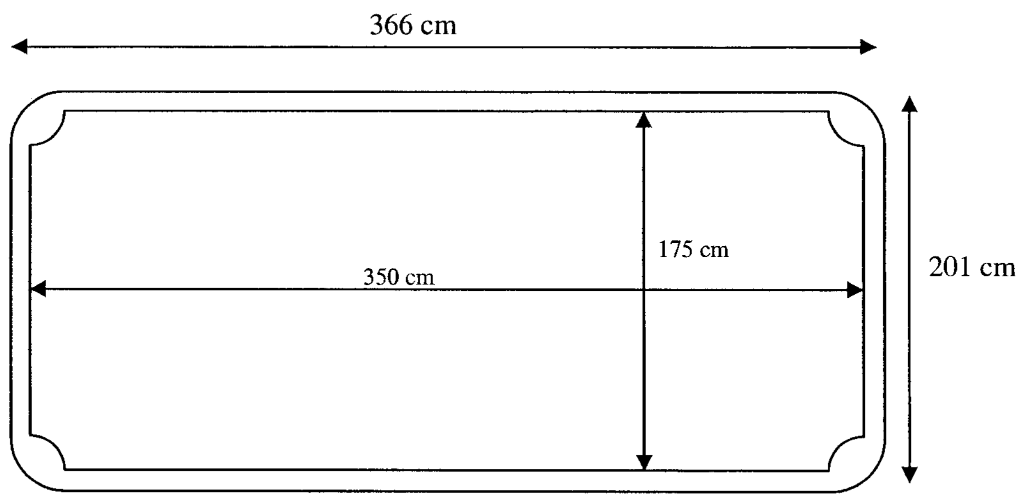


Figure 2.1-1 Scale of the snooker table

The table is consists of four edges. Each of them is covered by three different textures: Green cloth, dark brown wood and grey metal.

The height of the edge is 10cm.

The width of the edge is 13cm.

The pockets and other detailed graphics require detailed modeling with large amounts of coordinate data. For reasons of space, this report does not list all of the data, but it can be found in the source listing of the program.

2.2 Platform and Programming Language

Selecting a powerful API and a good programming language was the first part of the project.

OpenGL has a number of benefits for the application developer:

- **Reliable implementation:** Each implementation of OpenGL must adhere to the OpenGL specification and pass a set of conformance tests. However, implementations are not guaranteed to be identical at the pixel level; consequently an OpenGL program may give slightly different results on different platforms.

- **Platform independence:** It is a cross-platform system, allowing users to leverage user's development efforts for other platforms. OpenGL drivers free users from designing for specific hardware features and ensure consistent presentation on any compliant hardware/software configuration.

- **Industry acceptance**: In addition to OpenGL for SGI Workstations, there are OpenGL implementations for Windows, Linux, Irix, Solaris, BeOS, and game consoles.

- **Performance:** OpenGL uses available 3D acceleration hardware features to improve rendering speeds. OpenGL is designed in such a way that graphics cards can provide hardware acceleration. In other words, OpenGL does not define the precise boundary where software ends and hardware begins. This is one of its great strengths – an OpenGL program gets faster when the user updates the graphics card or drivers, without the need for software changes.
- **Controlled evolution**: The OpenGL extension mechanism allows users to take advantage of hardware innovations early. Successful innovations are then

incorporated into the core OpenGL API as appropriate. At the same time, the standard enforces backward compatibility to extend the usable life of the application.

- **Full feature set:** The Core OpenGL API includes over 250 graphics routines, providing geometric and raster primitives, math routines, display list or immediate mode rendering, and viewing and modeling transformations.

- **Efficiency:** OpenGL routines help to keep the application small because it typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages.[7]

Before programming, I decided to use the GL Utility Toolkit (GLUT) with OpenGL for this project based on the research I had completed previously. GLUT offers several advantages over the Windows API since it is cross platform capable, allowing code written on GLUT to run on almost any UNIX distribution and on Microsoft Windows. In addition, GLUT offers superior speed to the Windows API when running on comparable hardware under a UNIX distribution. GLUT also offers a full screen mode through the GLUT game mode. However, this command is complicated to work with and yields different results depending on the acceleration hardware and operating system used.

DirectX is another popular software development kit controlled by Microsoft. It supports many features; including 2D graphics, 3D graphics, sound, input devices, multiplayer support, etc. DirectX changes significantly with each new release; however, old functions are always supported for backwards compatibility. DirectX is only supported on Microsoft operating systems and the XBox. It is difficult to learn and master. The third-party documentation and tutorials are not always helpful. DirectX is supported by almost all major vendors. The majority of commercial games on the market are developed with DirectX; an important consideration if we want to make a career out of writing video games.

3 TECHNICAL BACKGROUND

3.1 3D graphics Fundamentals

Before starting the discussion of my project, I give the fundamental concepts of 3D graphics and coordinate system.

3.1.1 3D perception

"3D Computer graphics" actually provides a two-dimensional images on a flat computer screen with the illusion of depth, or a third "dimension". In order to truly see in 3D, people need to actually view the object with both eyes, or supply each eye with separate and unique images. Each eye receives a two-dimensional image that is much like a temporary photograph on the retina. These two images are slightly different because they are received at two different angles. The brain then combine these slightly different images to produce a single, composite 3 D picture in our head, as shown in ***Figure 3.1-1***.

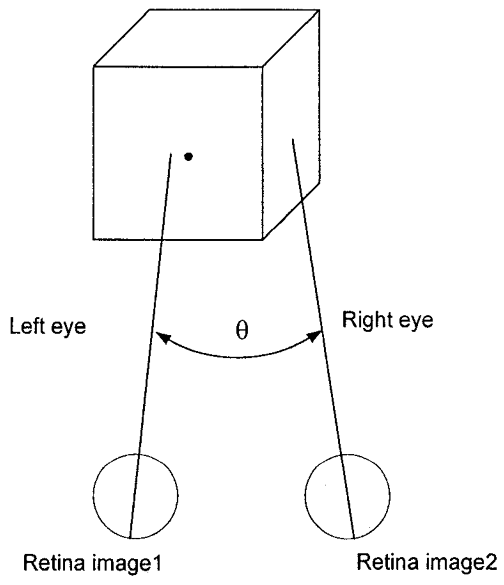


Figure 3.1-1 How the eye "sees" three dimensions

3.1.2 2D + Perspective = 3D

When people cover one eye, the world will not be suddenly flatten. This is because many of the 3D world's effects are also present in a 2D world. This is just enough to trigger our brain's ability to discern depth. The most obvious cue is that nearby objects appear larger than distant objects and parallel lines do not appear parallel, etc. This effect is called **perspective.** ***Figure 3.1-2*** presents a simple wire frame cube. The brain actually uses many techniques to distinguish depth, and integrates them all so smoothly that we are almost unaware of what it is doing. It uses direction for short distances, the size of known objects for medium distances, and clarity for large distance (hills appear distant because they are hazy). Strictly, perspective is a device used by artists to simulate depth rather than a mechanism in the brain.

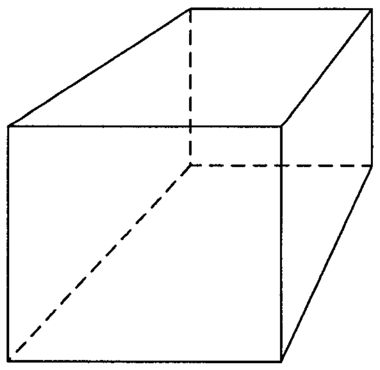


Figure 3.1-2 This simple wire frame cube demonstrates perspective

3.2 Graphic Coordinate system

Programs that display graphics on a computer screen have to deal extensively with a coordinate system similar to what we use when plotting functions in math classes. When producing graphic output on the drawing plane, we indicate where to place objects with coordinates. **Coordinates** are a pair of numbers that specify the *x* and *y* placement of a point. When a window is first created, the origin (that is, $x = 0$, $y = 0$) of the drawing plane is positioned at the top-left corner of the window.

There is one difference between the coordinate system typically used for computer graphics and those used in math classes. The y-axis in the coordinate system used in computer graphics is upside down. Thus, while our experience in algebra class might lead us to expect the point (2,3) to appear below the point (2,5) but, on a computer screen, just the opposite is true. This difference is illustrated by the ***Figure 3.2-1*** which shows where these two points fall in the normal Cartesian coordinate system and in the coordinate system used to specify positions when drawing on a computer screen.

Windows coordinates are upside-down. OpenGL, however, uses the mathematical convention: X right, Y up, Z towards us. The projection transformation performs the inversion.

Many programs may be running on a computer at once and each should only produce output in its own window. To make this simple, a distinct coordinate system is associated with each window. Each program's drawing commands are interpreted using its own personal coordinate system. This makes it possible for a program to produce graphical output without being aware of the location of its window relative to the screen boundaries or the locations of other windows.

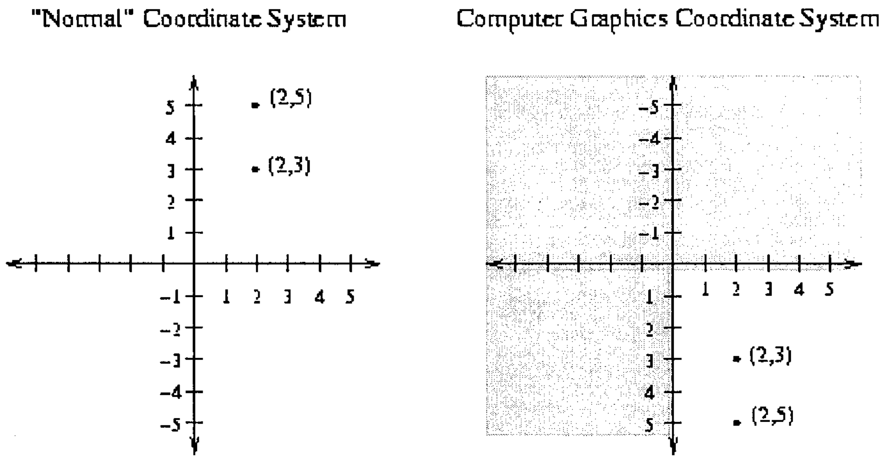

"Normal" Coordinate System
Computer Graphics Coordinate System
(2,5)
(2,3)
1 2 3 4 5
-1 -2 -3 -4 -5
(2,3)
(2,5)

Figure 3.2-1

In interpreting the graphic commands, the computer will assume that the interior of the program's screen window corresponds to the lower right quadrant of the computer graphics coordinate system shown above. That is, the coordinates of the upper-left hand corner of the window will be (0,0) and, if the window is 200 units wide and 300 units high, then the coordinates of the lower right corner will be (200,300). Anything we attempt to draw at coordinates that fall beyond the boundaries of the program's window will be ignored. Coordinates with negative values are always outside the window, no matter how large the window is. This is indicated by the gray background in the ***Figure 3.2-1.***

3.3 OpenGL Overview

In this section, I will introduce OpenGL. That includes what OpenGL is, OpenGL architecture, and the OpenGL model.

3.3.1 What is OpenGL?

OpenGL is an open, cross-platform three-dimensional (3D) graphics standard with broad industry support. OpenGL greatly eases the task of writing real-time 2D or 3D graphics applications by providing a mature, well-documented graphics processing pipeline that supports the abstraction of current and future hardware accelerators.

OpenGL was developed by Silicon Graphics, Inc. (SGI), based on SGI's IRIS GL (Graphics Library), first released in 1992. As an open standard, it is now controlled by the OpenGL Architecture Review Board (ARB), a consortium whose members represent many of the significant companies in the computer graphics industry.

OpenGL provides an interface to whatever 3D hardware is present on the machine, handling polygonal rendering, Z-buffering, texturing, and multiple light-sources. Complementing the rendering interface is GLUT, the GL Utility Toolkit, which can handle input from the keyboard and mouse, and idle function callbacks.

The key to understanding OpenGL is to know that it is a state machine. This means that we can put it into a certain state or mode, such as the color, position, or polygon drawing mode, which affects all subsequent operations until we change or remove it. The primary aspects of OpenGL are the buffers, the viewing and projection matrices, and the state variables.

In addition to an image buffer(frame buffer), OpenGL supports various types of ancillary buffers. For example, a window might also have a stencil buffer and a depth buffer. Modes such as stereo and double buffering are also supported. In double buffered mode, only one of the buffers is visible at a time. The currently visible buffer is referred to as the "front buffer" and the off screen buffer is referred to as the "back buffer". During the animation process, OpenGL drawing commands are directed to the hidden back buffer. Once a scene is fully rendered onto the back buffer, the front and back buffers are "swapped" so that the back buffer becomes visible (almost instantaneously) and the front buffer is hidden. Double buffering also avoids flickering and broken images. The key is that drawing and clearing operations never occur on a buffer while it is visible [5]. The depth buffer also called the **Z-buffer**, and most applications use it. The depth buffer doesn't store fragment colors. Instead, it stores their distance to the screen. If a fragment being written has a greater depth than the one already in the depth buffer, that fragment is invisible, and should not be drawn [6].

The viewing matrix is the transformation which is applied to everything rendered; it can be pushed and popped on a stack so that hierarchical structures can be built. The variables handle color, lighting information, and rendering styles. For more information on how OpenGL works, the OpenGL Programming Guide [1] is recommended.

3.3.2 The OpenGL Model

The diagram in ***Figure 3.3-1*** shows the basic components of OpenGL and their relation to the application and to each other.

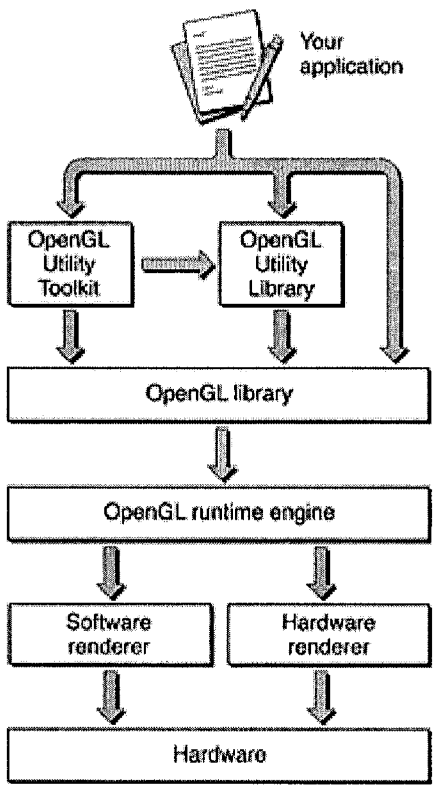


Figure 3.3-1 OpenGL Architecture

The current state is affected by issuing commands. OpenGL commands occur within the context of this current state. For example, to draw a red triangle, we would begin *triangle drawing mode*, set the color to red, specify three vertices, and then end *triangle drawing mode*.

The OpenGL API was designed with a client/server model in mind, as shown in

Figure 3.3-2. Commands are issued by the application, the client, and processed by the server. The server manages the frame buffer, the buffer that drawing commands are rendered into. Typically, the frame buffer is displayed onscreen by the client as a grid of pixels. Each channel of client/server communication contains its own OpenGL state. Because of the client/server abstraction, the core OpenGL API does not provide functions for context creation, frame buffer management, or event handling; these tasks are platform specific. However, the OpenGL Utility Toolkit provides a standard, cross-platform API for these operations.

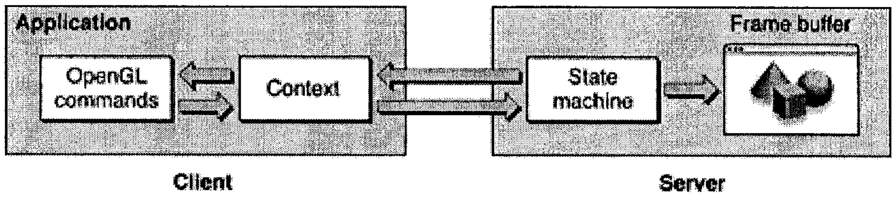


Figure 3.3-2 OpenGL Model

The application will typically interface directly with the core OpenGL library (GL), the OpenGL Utility library (GLU), and the OpenGL Utility Toolkit (GLUT).

GLU is a set of functions to create texture mipmaps from a base image, map coordinates between screens and object space, draw quadric surfaces and NURBS, and perform other tasks that are not implemented in the core library. GLUT is a window system-independent toolkit for writing OpenGL programs. It implements a simple windowing application programming interface for OpenGL. GLUT makes it considerably easier to learn about and explore OpenGL programming and provides a portable API.

GL is a low-level modular API that allows developers to define graphical objects. It includes the core functions that are common to all OpenGL implementations, as mandated by the OpenGL specification. It provides support for two fundamental types of graphics primitives: objects defined by a set of vertices, such as line segments and simple polygons, and objects that are pixel-based images, such as filled rectangles and bitmaps. Support for complex custom graphical objects is not provided by GL; the application must decompose them into simpler geometries.

3.4 Graphic Concepts

This section outlines some fundamental graphic concepts and explains how they are handled within OpenGL. Six models are discussed. The first is the color model, explaining how different colors are represented within the virtual environment. The second model regards matrices and explains positioning of 3D objects within the world. Thirdly, the viewing model is presented, explaining how the user view is achieved. Fourthly, the lighting model explains how the scene is lit. Fifthly, the curve and surface model explains how evaluators work and describes Bezier Curves and the GLU NURBS Interface. Finally a brief, broad look at the steps required to perform texture mapping is presented.

3.4.1 The Color Model

RGB Color Model

OpenGL uses the standard RGB color model based on the color cube (see ***Figure 3.4-1***). The RGB values are specified as real numbers between 0 and 1 (or whole numbers between 0 and 255 when dealing with textures). There is a fourth value or 'channel' commonly called the 'alpha channel' which represents the colors opacity. This is always specified as a number between 0 and 1. 0 represents full transparency and 1 signifies being fully opaque [1].

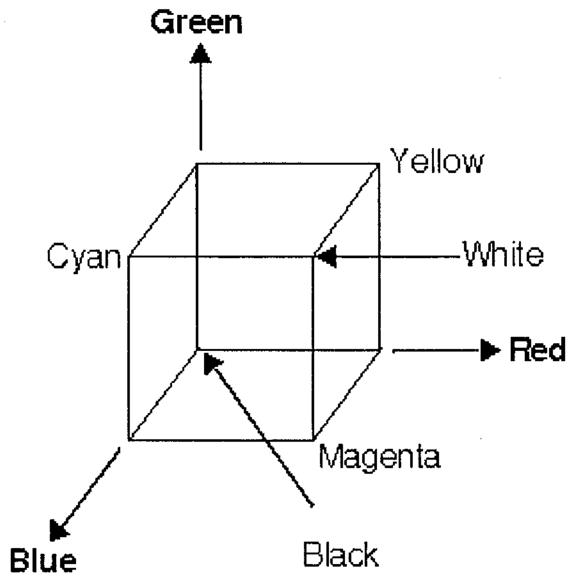


Figure 3.4-1 RBG color model

The RGB model is widely used but extremely device-dependent. When the device is changed, the color will change too. It is not suitable for color reproduction when a number of devices, such as a scanner, monitor, and printer must be used together. Since it uses the three additive primary colors, it is not suitable for paints or for the dyes and pigments used in printing, which use a different set of primary colors (cyan, magenta, yellow). The usual set of printing primaries includes black (and is called "CMYK") because it is wasteful to print black using three colored inks.

CIE Color Model

These color models were created by the Commission Internationale de l'Eclairage/International Commission on Illumination (CIE). They are based on the human eye's response to RGB, and are designed to accurately represent human color perception.

These models are used to define so-called device-independent colors, which can be reproduced faithfully on any type of device, such as scanners, monitors, and printers. They are widely used because they are easy to use on computers and describe a wide range of colors.

The best known models are the CIE XYZ and CIE L*a*b.

CIE XYZ

In 1931 the CIE developed the XYZ color system, also called the "norm color system". This system is often represented as a two-dimensional graphic which more or less corresponds to the shape of a sail.

The red components of a color are tallied along the x (horizontal) axis of the coordinate plane and the green components along the y (vertical) axis. In this way every color can be assigned a particular point on the coordinate plane. The spectral purity of colors decreases as moving left along the coordinate plane. What is not taken into consideration in this model is brightness.

CIE L*a*b*

CIE L*a*b is an improvement of the CIE XYZ color model. In this three dimensional model, the color differences which we perceive correspond to distances when measured calorimetrically. The a axis extends from green (-a) to red (+a) and the b axis from blue (-b) to yellow (+b). The brightness (L) increases from the bottom to the top of the three-dimensional model. Colors are represented by numerical values. In comparison with XYZ, CIE L*a*b* colors are more compatible with colors sensed by the human eye. With the CIE L*a*b*, the color luminance (L), hue and saturation (a, b) can be revised individually; as a result, the overall color of the image can be changed without changing the image or its luminance. Because CIE L*a*b* is device independent, when RGB is changed to CMYK, or CMYK is changed to RGB, the software requires the change to be first processed via the CIE L*a*b* color model [8].

3.4.2 The Matrix Model

In OpenGL, all vertices are transformed by two matrices. These are the **modelview matrix** and the **projection matrix**. The modelview transformation precedes the projection transformation. The projection matrix is responsible for defining a view volume and clipping to this volume. So it is common in OpenGL application to setup the projection matrix once and leave it alone afterwards.

In any case, here's a 4x4 matrix.

[A1 A2 A3 A4]
[B1 B2 B3 B4]
[C1 C2 C3 C4]
[D1 D2 D3 D4]

A matrix is a two dimensional array of numeric data, where each row or column consists of one or more numeric values. Arithmetic operations which can be performed with matrices include addition, subtraction, multiplication and division. The size of a matrix is defined in terms of the number of rows and columns. A matrix with M rows and N columns is called a MxN matrix. In C++ a 4x4 matrix can be stored as an array of Matrix[4][4]; or Matrix[16];

We can use this transformation system is used to transform 3D geometry translate, rotate and scale 3D points and vectors.

Identity

First take a look at this 4x4 homogeneous identity matrix. The identity matrix is matrix in which has an identical number of rows and columns. Also, all the elements in which i=j are set one. All others are set to zero. For example a 4x4 identity matrix is as follows:

```
[ 1  0  0   0  ]
[ 0  1  0   0  ]
[ 0  0  1   0  ]
[ 0  0  0   0  ]
```

Translate

A translation matrix is used to position an object within 3D space without rotating in any way. Translation operations using matrix multiplication can only be performed using 4x4 matrices. If the translation is defined by the vector [X Y Z], then the 4x4 matrix to implement translation is as follows:

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If the vector is [0 0 0] then the vertex list will remain as before.

Scale

A scaling matrix is used to enlarge or shrink the size of a 3D model. If the scaling vector is [X Y Z] then the matrix to perform this is as follows:

$$\begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

If the scaling vector is [1 1 1], then this generates the identity matrix and vertex geometry will remain unchanged.

3.4.3 The Viewing Model

By default, OpenGL uses orthographic projection, which maps objects directly onto the screen without affecting their relative sizes. This projection is used mainly in architectural and computer-aided design applications, where the actual measurements of the objects are more important than how they might look. The command *glMatrixMode()* is used with the argument GL_PROJECTION to set up the matrix upon which this projection transformation (and subsequent transformations) is performed.

glLoadIdentity() command is used to initialize the current projection matrix to the identity matrix so that only the specified projection transformation(s) have an effect. Finally, the command *glOrtho()* is called to create an orthographic parallel viewing volume. The viewing volume is a box with the left, right, top, bottom, near and far edges specified in the *glOrtho()* command.

The following short code example will give us a better idea of how it works.

```
glMatrixMode(GL_PROJECTION);
glPushMatrix();              /*save the current projection*/
  glLoadIdentity();
  glOrtho(...);              /*set up for displaying help*/
  display_the_help();
glPopMatrix();
```

A perspective projection will provide a more realistic rendering of an object. The object(s) will have the unmistakable characteristic of foreshortening: the further an object is from the camera, the smaller it appears in the final image. This is because the viewing volume of perspective projection is a frustum (a truncated pyramid whose top has been cut off by a plane parallel to its base). Objects that are closer to the apex of the pyramid appear smaller while objects closer to the base appear larger. The command that defines the frustrum is *glFrustum()*, which takes the values for left, right, top, bottom, near and far edges in following code example. It is possible to perform rotations or translations on this projection matrix to alter its orientation, but these are tricky and should be avoided [1].

```
glMatrixMode (GL_PROJECTION);          /* prepare for and then */
   glLoadIdentity ();                  /* define the projection */
```

```
    glFrustum (-1.0, 1.0, -1.0, 1.0,            /* transformation */
              1.5, 20.0);
glMatrixMode (GL_MODELVIEW);                    /* back to modelview matrix */
glViewport (0, 0, w, h);                        /* define the viewport */
```

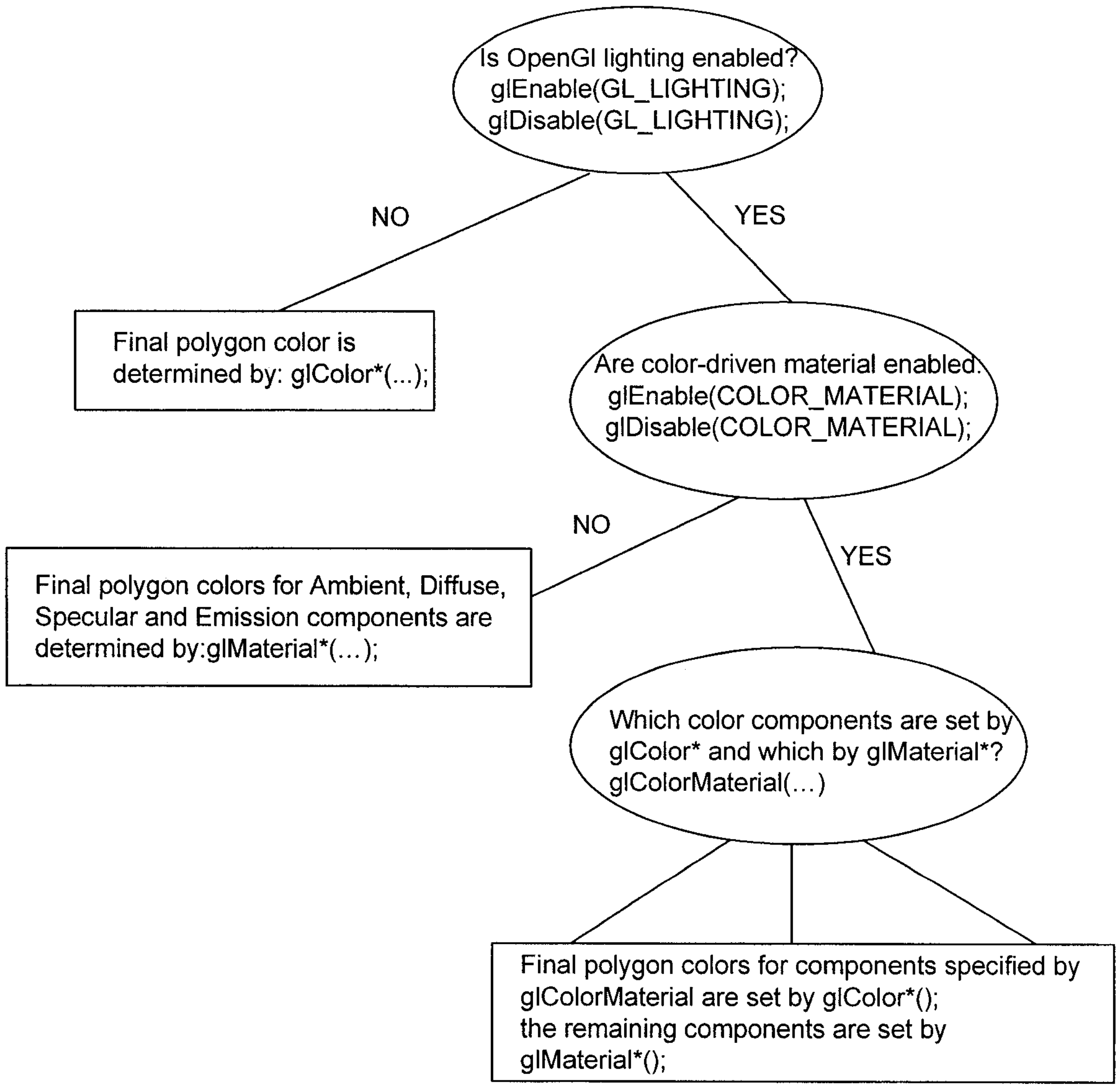
3.4.4 The Lighting Model and Surface Properties

A world with no light is entirely dark, and the user is presented with a black screen. Thus, a light source (or light sources) must be present within the world in order to view the objects it contains. In the OpenGL lighting model, the light in a scene comes from several light sources that can individually be turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene.

In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (like headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or shiny surface.

The OpenGL lighting model considers the lighting to be divided into four independent components: emitted, ambient, diffuse, and specular. All four components are computed independently, and then added together.

The flow chart below is clearly explaining the way the OpenGL's built in lighting works:


Is OpenGl lighting enabled?
glEnable(GL_LIGHTING);
glDisable(GL_LIGHTING);
NO
YES
Final polygon color is determined by: glColor*(...);
Are color-driven material enabled?
glEnable(COLOR_MATERIAL);
glDisable(COLOR_MATERIAL);
NO
YES
Final polygon colors for Ambient, Diffuse, Specular and Emission components are determined by:glMaterial*(...);
Which color components are set by glColor* and which by glMaterial*?
glColorMaterial(...)
Final polygon colors for components specified by glColorMaterial are set by glColor*();
the remaining components are set by glMaterial*();

3.4.5 Curves and Surface model

At the lowest level, graphics hardware draws points, line segments, and polygons, which are usually triangles and quadrilaterals. Smooth curves and surfaces are drawn by approximating them with large numbers of small line segments or polygons. However, many useful curves and surfaces can be described mathematically by a small number of parameters such as a few *control points*. Saving the sixteen control points for a surface requires much less storage than saving 1000 triangles together with the normal vector information at each vertex. In addition, the 1000 triangles only approximate the true surface, but the control points can accurately describe the real surface.

OpenGL supports curves and surfaces through mechanisms, called **evaluators**, that provide a way to specify points on a curve or surface (or part of one) using only the control points. The curve or surface can then be rendered at any precision. In addition, normal vectors can be calculated for surfaces automatically. The points generated by an evaluator in many ways are used to draw dots where the surface would be, to draw a wire frame version of the surface, or to draw a fully lighted and shaded version.

Evaluators are used to describe any polynomial or rational polynomial splines or surfaces of any degree. These include almost all splines and spline surfaces in use today, including B-splines, NURBS (Non-Uniform Rational B-Spline) surfaces, Bézier curves and surfaces, and Hermite splines. Since evaluators provide only a low-level description of the points on a curve or surface, however, they are typically hidden within utility libraries that provide a higher-level interface to the programmer. The GLU's NURBS facility is such a higher-level interface — the NURBS routines encapsulate lots of complicated code, but the final rendering is done with evaluators.

Bezier Curves

A Bézier curve is a vector-valued function of one variable

$\mathbf{C}(u) = [\mathbf{X}(u)\ \mathbf{Y}(u)\ \mathbf{Z}(u)]$

where u varies in some domain (say [0,1]). A Bézier surface patch is a vector-valued function of two variables

$\mathbf{S}(u,v) = [\mathbf{X}(u,v)\ \mathbf{Y}(u,v)\ \mathbf{Z}(u,v)]$

where u and v can both vary in some domain. The range isn't necessarily three-dimensional as shown here. It could be two-dimensional output for curves on a plane or texture coordinates, or four-dimensional output to specify RGBA information. Even one-dimensional output may make sense for gray levels. [1]

The GLU NURBS Interface

Although evaluators are the only OpenGL primitive available to directly draw curves and surfaces, and even though they can be implemented very efficiently in hardware, they are often accessed by applications through higher-level libraries. The GLU provides a NURBS (Non-Uniform Rational B-Spline) interface built on top of the OpenGL evaluator commands.

According to earlier observation, it is possible to change any of the polynomial forms to a Bezier form by generating a proper set of control points, and then all the information necessary to define NURBS curves and surfaces are available. The GLU provides a set of

NURBS functions that allow the user to specify varies additional parameters that enable finer control of the rendering. There are five NURBS functions for surfaces:

gluNewNurbsRenderer

gluNurbsProperty

gluBeginNurbsSurface

gluNurbsSurface

gluEndNurbsSurface

The first two functions set up a new NURBS object and define how developers would like it rendered. The final three are used to generate the surface. For NURBS curves, the final three are replaced by the functions *gluBeginNurbsCurve, gluNurbsCurve,* and *gluEndNurbsCurve* [1].

3.4.6 Texture mapping

Texture mapping is one of the most successful new techniques in high quality image synthesis. Its use can enhance the visual richness of raster scan images immensely while entailing only a relatively small increase in computation. The technique has been applied to a number of surface attributes: surface color, surface normal, specularity, transparency, illumination, and surface displacement, to name a few. Although the list is potentially endless, the techniques of texture mapping are essentially the same in all cases.

A "texture" can be either a texture in the usual sense (e.g. cloth, wood, gravel) – a detailed pattern that is repeated many times to tile the plane, or more generally, a multidimensional image that is mapped to a multidimensional space. Texture mapping means the mapping of a function onto a surface in 3-D. The domain of the function can be one, two, or three-dimensional, and it can be represented by either an array or by a mathematical function. The source image (texture) is mapped onto a surface in 3-D object space, which is then mapped to the destination image (screen) by the viewing projection. Texture space is labeled (u, v), object space is labeled (x_o, y_o, z_o), and screen space is labeled (x, y).

In order to upload textures, a call to *glBindTexture* call is the first thing that is required. It tells OpenGL which texture "id" we will be working with. A texture "id" is just a number that will be used to access textures. Any calls that have to do with OpenGL texture mapping will affect this texture. The program must store this number since it will be needed again later on to actually apply the texture to geometry.

The glPixelStorei call tells OpenGL how the data that is going to be uploaded is aligned. A call to ***glPixelStorei*** is shown below.

glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

This call tells OpenGL that the pixel data which is going to be passed to it is aligned in byte order, this means that the data has one byte for each component, one for red, green and blue.

The *glTexParameteri* call sets various parameters for the current OpenGL texture. The parameters that are passed and their effects on the texture are an advanced topic. It is important to get following lines in the application.

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri (GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
```

The *glTexEnvf* call sets environment variables for the current texture. It tells OpenGL how the texture will act when it is rendered into a scene. Below is a sample call which I use in my applications.

```
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
```

The GL_REPLACE attribute allows texture color to replace object color.

The *glTexImage2D* call is the goal. This call will upload the texture to the video memory where it will be ready to use in programs.

4 PROJECT IMPLEMENTATION

In this project, there are many curve surfaces need to be simulated. For example, pockets, the area near pockets, and the corners of the table. The method I used for generating curved surfaces is to divide them to several sections, and then subdivided each to many tiny polygons. This process is called "surface approximation”. The denser the mesh, the better is the approximation.

4.1 Create an OpenGL window

Before displaying anything with OpenGL, a display window must be created. In this project the following code is used to build the window.

Here I create an OpenGL display window with following characteristics:

- Double buffer: To avoid flicker, I set the window with attribute of double buffer, thus OpenGL renders the image into one buffer while displaying the contents of the other buffer.
- Depth buffer: by using the depth buffer, I eliminate hidden surfaces. That means that OpenGL displays only the objects that the viewer can see.
- Use RBG color mode

The initial window size is 1200 x 1200, and its initial window position is at (50 x 50).

The intent of the initial window position and size values is to provide a suggestion to the window system for a window's initial size and position. In the program, I use the window's reshape callback to determine the true size of the windows if the screen is not that big.

Code example :

```
void setFrustum(void){
 glMatrixMode(GL_PROJECTION);
 glLoadIdentity();
 glFrustum(-width, width, - height, height, 1.0, 20.0);
 glMatrixMode(GL_MODELVIEW);
}
void reshape(int w, int h) {
 glViewport(0, 0,  w,  h );
 width = height * (GLfloat)w/(GLfloat)h;
 setFrustum();
}
int main(int argc, char** argv){
        glutInit( &argc, argv);
        glutInitDisplayMode(GLUT_DOUBLE|GLUT_RGB|GLUT_DEPTH);
        glutInitWindowSize(SIZE,SIZE);
        glutInitWindowPosition(50,50);
        glutCreateWindow(argv[0]);
        init();
        InitTextures();
        glutDisplayFunc(display);
        glutReshapeFunc(reshape);
        glEnable(GL_DEPTH_TEST);
        glutMainLoop();

        return 0;

}
```

4.2 Model the tabletop

From the view of the table, the tabletop looks like the regular rectangle. Actually, it's not. As the ***Figure 4.2-1*** shows, the tabletop has six arcs near the pockets. Then I divided the tabletop into thirteen sections.

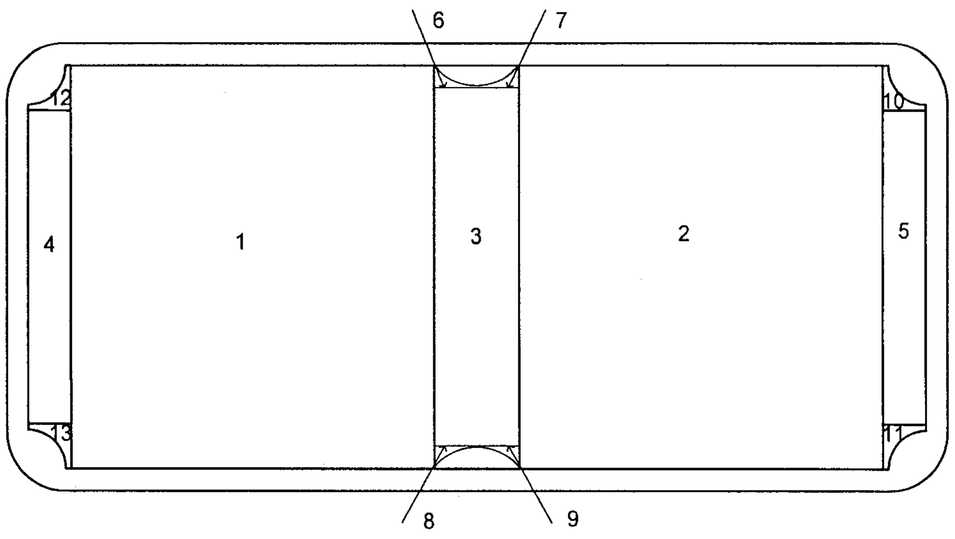


Figure 4.2-1 Thirteen sections of tabletop

Some sections of the tabletop are trivial to model, especially due to their rotational symmetry. They are built out of two OpenGL primitives: the quad strip and the triangle fan. A quad strip is a series of quads (polygons with four vertices), with adjacent quads sharing two points. A triangle fan is a series of triangles, all sharing one central point, and with adjacent triangles sharing one more point which is not the central one, as the ***Figure 4.2-2*** shows.

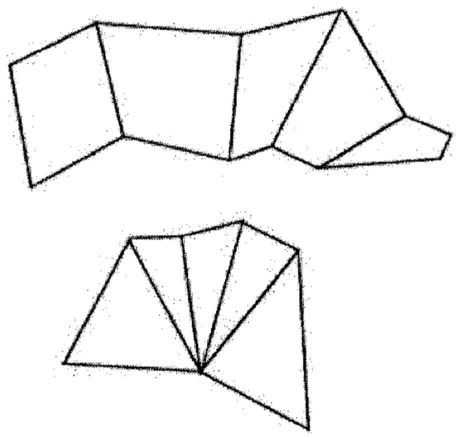


Figure 4.2-2 A quad trip and triangle fan

Code example:

```
glColor3f(0.0,1.0,0.0);
glBegin(GL_TRIANGLES);
   glVertex3f(TableCoverX[2],0.0,TableCoverY[0]);
   glVertex3f(NearRoleCoverX[0],0.0,TableCoverY[4]);
   glVertex3f(TableCoverX[2],0.0,TableCoverY[2]);
glEnd();


glColor3f(0.0,1.0,0.0);
glBegin(GL_POLYGON);
   glVertex3f(NearRoleCoverX[0],0.0,TableCoverY[4]);
   glVertex3f(NearRoleCoverX[2],0.0,NearRoleCoverY[0]);
```

```
    glVertex3f(TableCoverX[2],0.0,TableCoverY[2]);
glEnd();


glColor3f(0.0,1.0,0.0);
glBegin(GL_POLYGON);
    glVertex3f(NearRoleCoverX[2],0.0,NearRoleCoverY[0]);
    glVertex3f(NearRoleCoverX[4],0.0,NearRoleCoverY[2]);
    glVertex3f(TableCoverX[2],0.0,TableCoverY[2]);
glEnd();
```

A polygon has two sides - front and back - and might be rendered differently depending on which side is facing the viewer. By convention, polygons whose vertices appear in counterclockwise order on the screen are called front-facing. By default, *mode* is GL_CCW, which corresponds to a counterclockwise orientation of the ordered vertices of a projected polygon in window coordinates. If *mode* is GL_CW, faces with a clockwise orientation are considered front-facing [1].

4.3 Creation of Pockets

For the four pockets on table corners, I separate them into six sections, as the ***Figure 4.3-1*** shows.

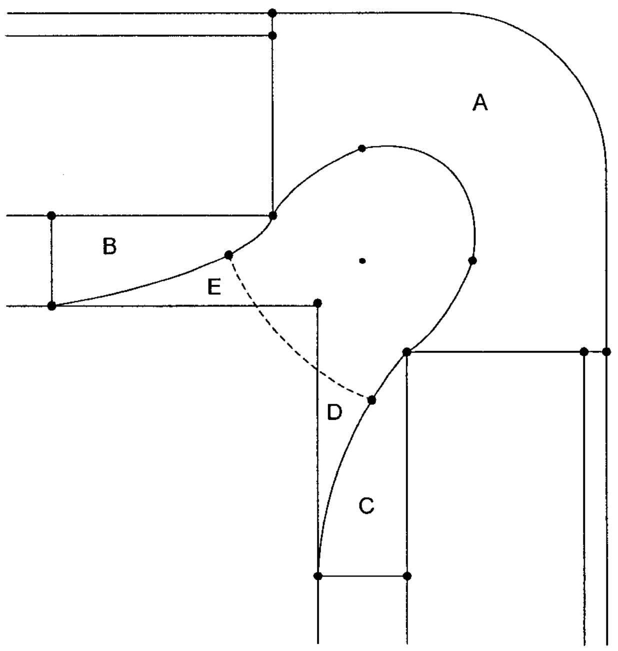


Figure 4.3-1 The scale of pocket on table corner

“A” section is mapped by metal texture.

“B”, “C”, “D”, “E” sections are mapped by green table cover.

The result is as the ***Figure 4.3-2*** shows.

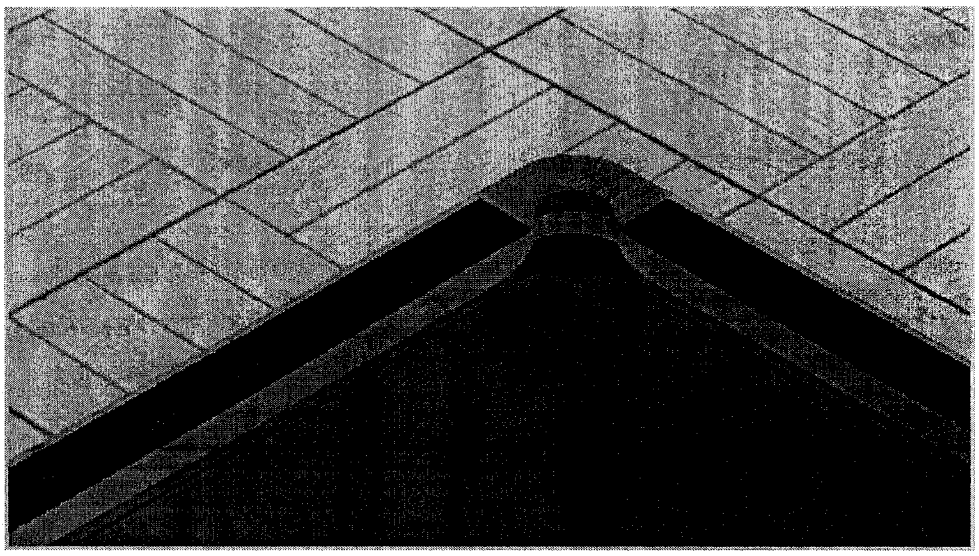


Figure 4.3-2 Close-up view of corner pocket drawing

For the pockets in the middle, I devided it into six coordinate systems, as the ***Figure 4.3-3*** shows:

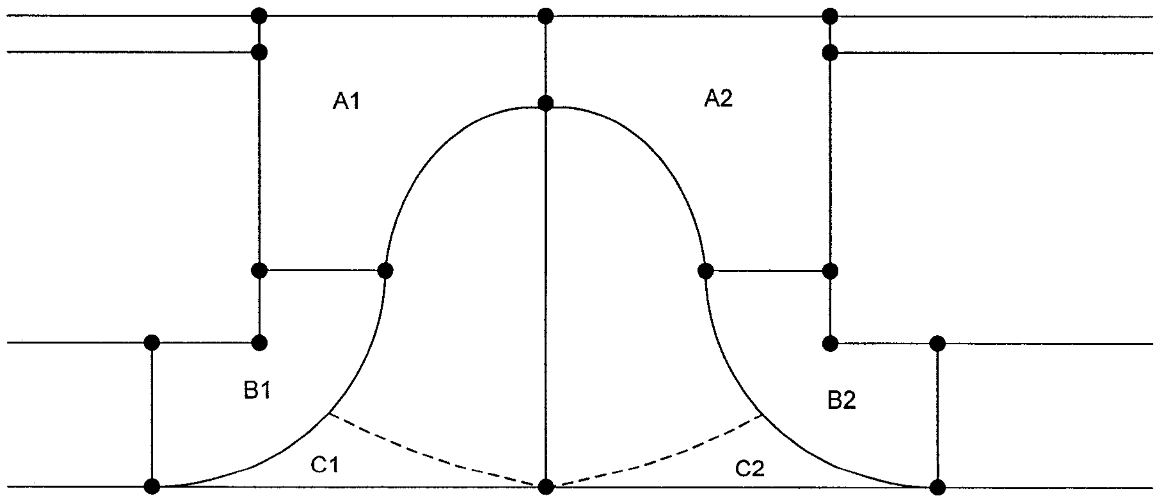


Figure 4.3-3 six sections of middle pockets

A1,A2 sections are texture mapped by grey metal.

B1,B2,C1,C2 sections are all texture mapped by green cloth.

Each section is drawn by a number of polygons.

The result is as the ***Figure 4.3-4*** shows.

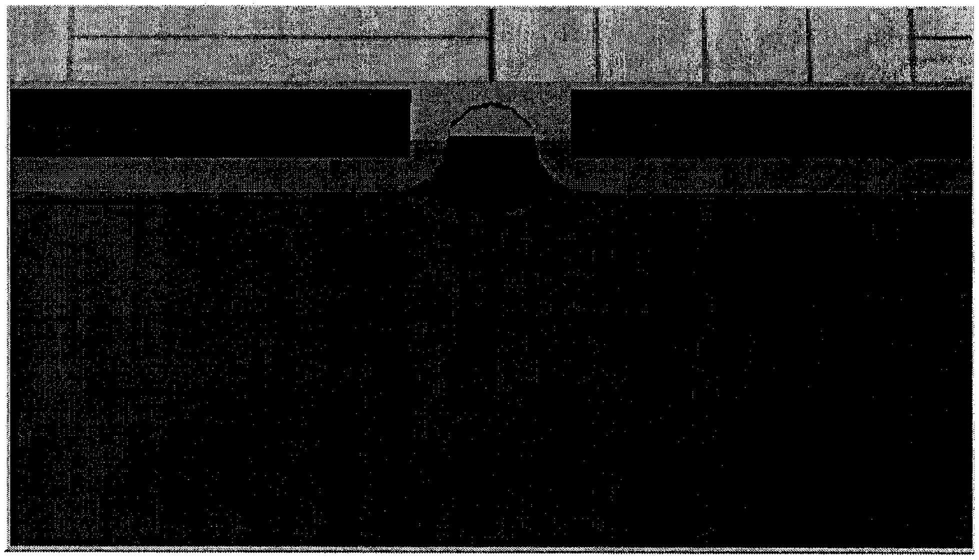


Figure 4.3-4 Close-up view of middle pockets

4.4 Table Edge Generation

The snooker table edge is composed of three parts covered by different textures, as ***Figure 4.4-1*** shows.

- green cloth (width = 4cm)
- dark brown wood (width = 8cm)
- grey metal (width = 1cm)

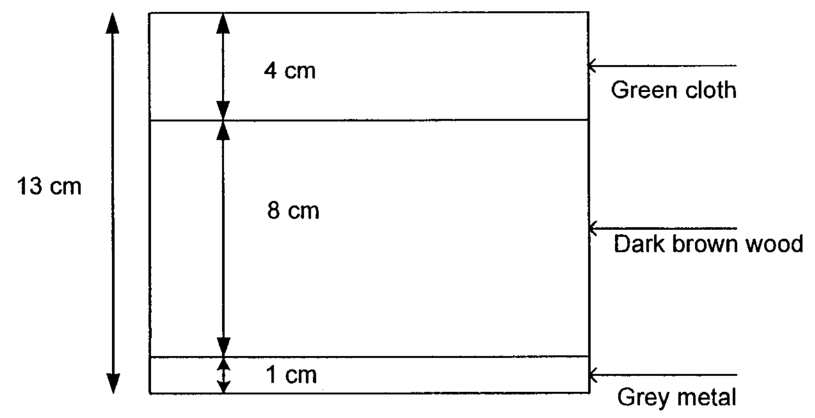


Figure4.4-1 Scale of the table edge

For the green cloth section, it is in a shape as ***Figure 4.4-2*** shows.

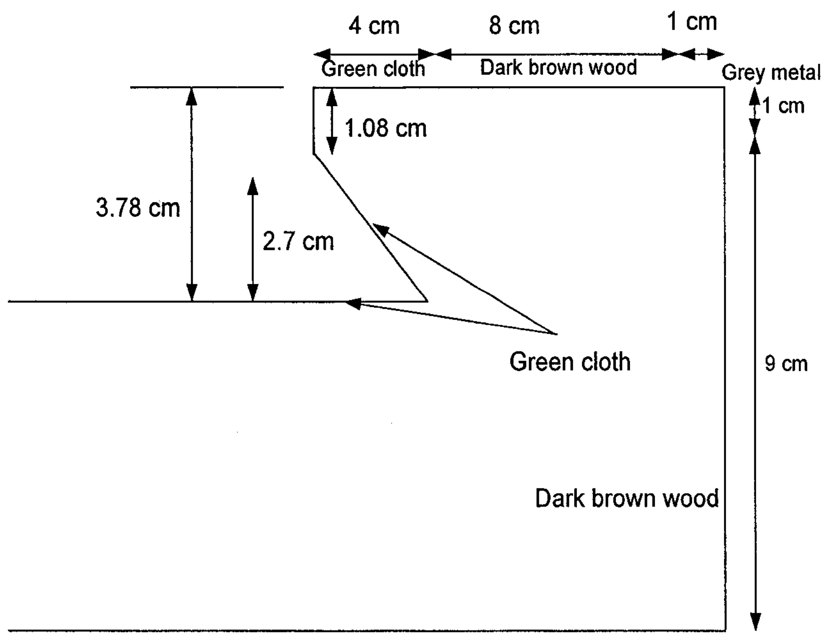


Figure 4.4-2 Green cloth section of edges

The edge is in a shape of regular rectangle, I just divided the longer edge to two sections: one is on the right side of middle pockets; one is on left side of middle pocket. The

Figure 4.4-3 shows the result of drawing;

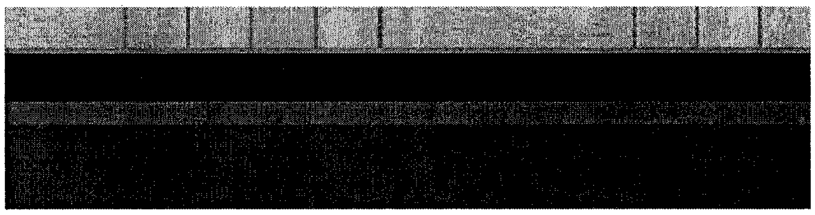


Figure 4.4-3 Result of edge drawing

Code Example:

```
void InitTextures(void)
{
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
        glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
Tex1.LoadFromFile("tablecloth3.bmp");
Tex2.LoadFromFile("tablewood2.bmp");
Tex3.LoadFromFile("tablemetal2.bmp");
Tex4.LoadFromFile("tablecloth4.bmp");
Tex5.LoadFromFile("Floor.bmp");
glTexEnvf(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE);
}
Tex3.SetActive();
glBegin(GL_POLYGON);
   glTexCoord2f(0.0f, 0.0f);
   glVertex3f(NearRoleCoverX[28],NearRoleCoverZ[2],NearRoleCoverY[39]);
```

```
    glTexCoord2f(0.2f, 0.0f);
    glVertex3f(NearRoleCoverX[20],NearRoleCoverZ[2],NearRoleCoverY[39]);
    glTexCoord2f(0.2f, 0.2f);
    glVertex3f(NearRoleCoverX[20],NearRoleCoverZ[3],NearRoleCoverY[39]);
    glTexCoord2f(0.0f, 0.2f);
    glVertex3f(NearRoleCoverX[28],NearRoleCoverZ[3],NearRoleCoverY[39]);
glEnd()
```

4.5 Viewpoints movement

In this project, I have implemented following functions:

- Rotating the table about x, y and z axis
- Close-up views for the pockets
- Moving views of the table
- Zooming in and zooming out

The main idea of rotation is get the current model view matrix and multiply others on the right because I want to rotate in the eye coordinate. I handle zoom in and zoom out by changing the width and height of the view column. For the close-up views and function of moving views, I use gluLookAt() to change the eye position.

To view the model from any angle requires a simple camera system. OpenGL handles camera movement by moving the scene relative to the viewer. The camera always points directly towards the centre of the table, but can be zoomed in and out (by translating closer and further from the centre), and rotated around both the y and x axes (by using OpenGL's rotate functions). The following sequence of commands demonstrates how the camera is positioned.

glLoadIdentity ();	Always starting with the identity viewing matrix
glTranslatef (0, 0, -distance);	Bring the camera out to the desired distance from the centre
glRotatef(x_angle, 1.0, 0.0, 0.0);	This rotates the camera around x-axis to look up or down towards the table
glRotatef(y_angle, 0.0, 1.0, 0.0);	This rotates the camera around the y axis to observe different sides of the snooker table
gluLookAt(0,dist,0.0, 0.0,0.0,0.0, 0.0,0.0,-1);	This creates a viewing matrix derived from eye point, a reference point indicating the center of the scene, and an UP vector

Because I am conceptually moving the scene rather than the camera, the positioning translations and rotations must be in reverse order. The resulting overall matrix will be applied to all primitives rendered, and so the table will appear in the correct position.

For ease of operating the applications, I use a group of hot-keys instead of menus. GLUT provides a callback function *glutKeyboardfunc(keyboard)* to define hotkeys. Once the program has registered a keyboard callback function, this function would be called whenever a key is pressed.

It would be possible to rewrite keyboard() so that it called glutPostRedisplay() once only, after the switch statement. The advantage of the code shown is that the scene is redisplayed only when one of the keys x, y, or z has been pressed; other keys are ignored.

I also use mouse button to change the views. The initial view is as ***Figure 4.5-1*** shows

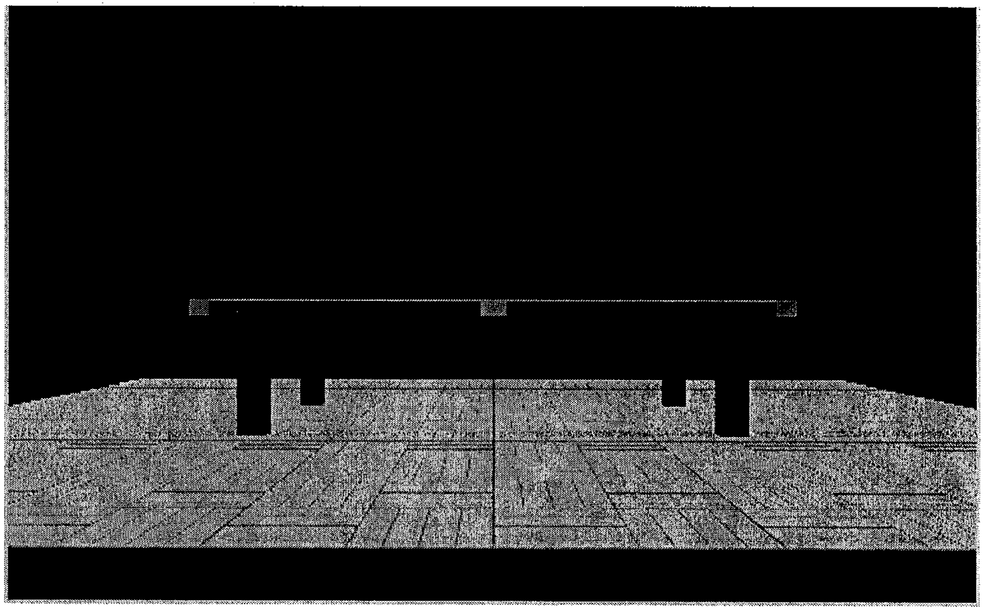


Figure 4.5-1 Initial viewing of the table

After clicking the right button of mouse once, the viewing will become as shown in *Figure 4.5-2*.

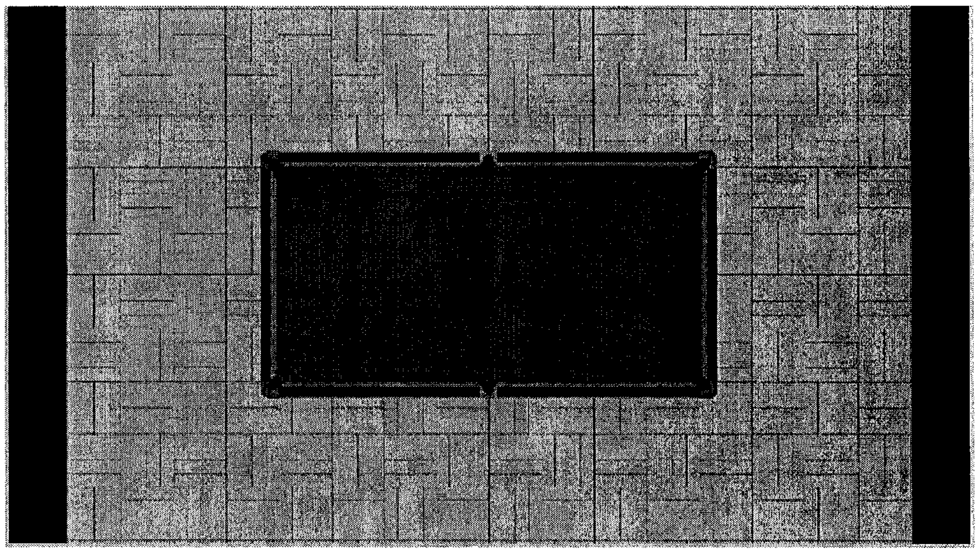


Figure 4.5-2 Viewing from Y axis

The function of moving views is implemented by clicking the right mouse button twice. This gives the impressions that the user is walking around the table. In the process of moving, there will be no response for any key pressing or button clicking. The height of eye position can be increased by pressing "F11", and be decreased by pressing "F12".

Code Example

```
void mouse_button(int button,int state, int x, int y)
{
 if(button == GLUT_RIGHT_BUTTON && state == GLUT_DOWN)
  {
        if(iLookPoint>1||iLookPoint<0)
        {
    iLookPoint = 0;
        glutPostRedisplay();
        }
        else if(iLookPoint==0)
        {
              iLookPoint = 1;
              glutPostRedisplay();
        }
        else
        {
              iLookPoint = -1;
             double degree = 0;
             double M_PI = 3.1415926;
              for(degree = 0;degree < 2*M_PI; degree= degree + M_PI/360)
              {
                   positionX = pZ * sin( degree );
                   positionZ = pZ * cos( degree );
                   display();
              }
             positionX = pX;
             positionZ = pZ;
             display();
        }
   }
}
```

After pressing the “-” button, the table will zoom out as ***Figure 4.5-3*** shows.

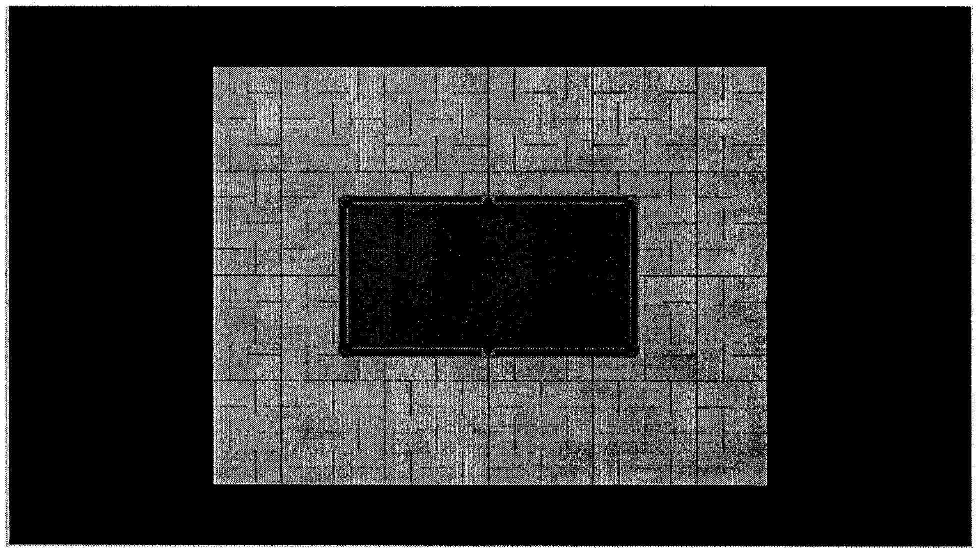


Figure 4.5-3 Viewing of zooming out

After pressing the “+” button, the table will zoom in as ***Figure 4.5-4*** shows.

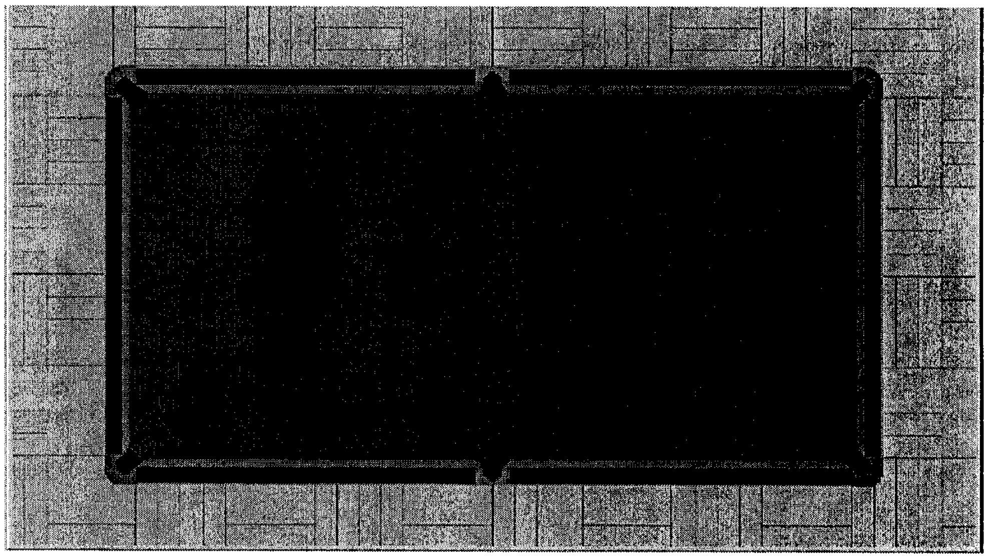


Figure4.5-4 Viewing of zooming in

After pressing the “↑”, “↓” keys, the table will rotates about the x axis. After pressing“←”, “→” keys, the table will rotates about the y axis. After pressing “Page Up”, “Page Down” keys, the table will rotates about the z axis.

For the close-up view of the pockets, just press " F1", "F3", "F4", "F6" keys, the views will be shown as ***Figure 4.3-2***(corner pockets). After pressing "F2", "F5" keys, the views will be shown as ***Figure 4.3-2***(middle pockets). Actually, the views of four corner pockets look the same instead of the changing of textures, and so are the views of two middle pockets.

Code Example:

```
void display(void) {
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glPushMatrix();
 switch(iLookPoint)
        {
        case 0:
          setFrustum();
          gluLookAt(0,0.0,dist,0.0,0.0,0.0,0.0,1.0,0);
         break;
case 1:
        setFrustum();
          gluLookAt(0,dist,0.0,0.0,0.0,0.0,0.0,0.0,-1);
          break;
case 3: //middle pocket on +z


          width /= 5;
          height /= 5;
          setFrustum();
```

```
                gluLookAt(0,0.5,0.5, 0.0,0.0,0.8, 0.0,0.0,1.0);
                width *= 5;
                height *= 5;
                break;
case 4: //corner pocket in (x, -z)
                width /= 5;
                height /= 5;
               setFrustum();
               gluLookAt(1.3,0.5,-0.3,1.8,0.0,-0.8,0.0,1.0,0.0);
               width *= 5;
               height *= 5;
              break;
case 7: //corner pocket in (-x, z)
                width /= 5;
                height /= 5;
                setFrustum();
                gluLookAt(-1.3,0.5,0.3,-1.8,0.0,0.8,0.0,1.0,0.0);
                width *= 5;
                height *= 5;
                break;
default:
                break;
               }
```

4.6 Lighting

In this project, I only have a positional light source act as a spotlight, which is characterized by a narrow range of angles through which light is emitted. A simple spotlight can be constructed from a point source by limiting the angles at which light from the source can be seen. A cone can be used whose apex is at **P**, which points in the direction **l**, and whose width is determined by an angle θ, as shown in ***Figure 4.6-1***. If θ = 180, the spotlight becomes a point source.

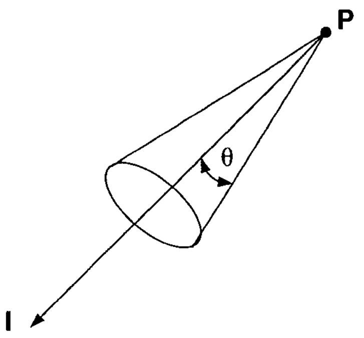


Figure 4.6-1 Spotlight

More realistic spotlights are characterized by the distribution of light within the cone – usually with most of the light concentrated in the center of the cone. Thus, the intensity is a function of angle Ø between the direction of the source and a vector **s** to a point on the surface (as long as this angle is less than 90 degrees; see ***Figure 4.6-1***). Although this function could be defined in many ways, it is usually defined by the cosine function, where the exponent (***Figure 4.6-2***, ***Figure 4.6-3***) determines how rapidly the light intensity drops off. Cosines are convenient functions for lighting calculations. If both **s** and **l** are unit-length vectors, we can compute the cosine with the dot product

$\cos Ø = \mathbf{s} \cdot \mathbf{l}$,

A calculation requires only three multiplications and two additions.

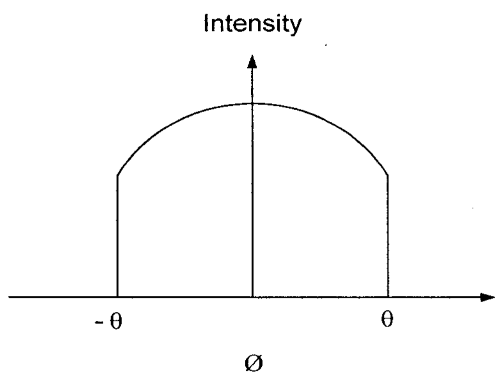


Figure 4.6-2: Attenuation of spotlight

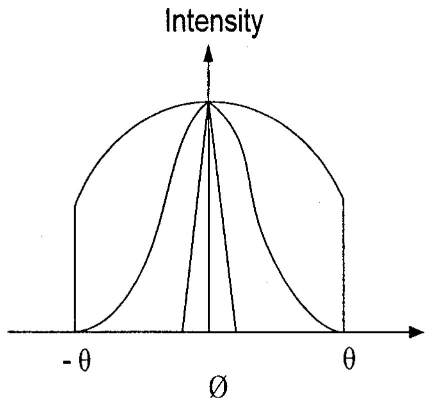


Figure 4.6-3 Spotlight exponent

However, using fancy light parameters requires many per-pixel calculations and slows down the display. This could be important for a game with animation [2].

4.7 Shadow of the table edge

I draw the shadows of the table edges, which are created by the spotlight. As the ***Figure 4.7-1*** shows, the shadow is very small. Then, it is hard to recognize it from the viewing.

I calculated the matrixes for shadowing an object to the table cover by the light of the spotlight. Then I drew the shadow by using those matrixes.

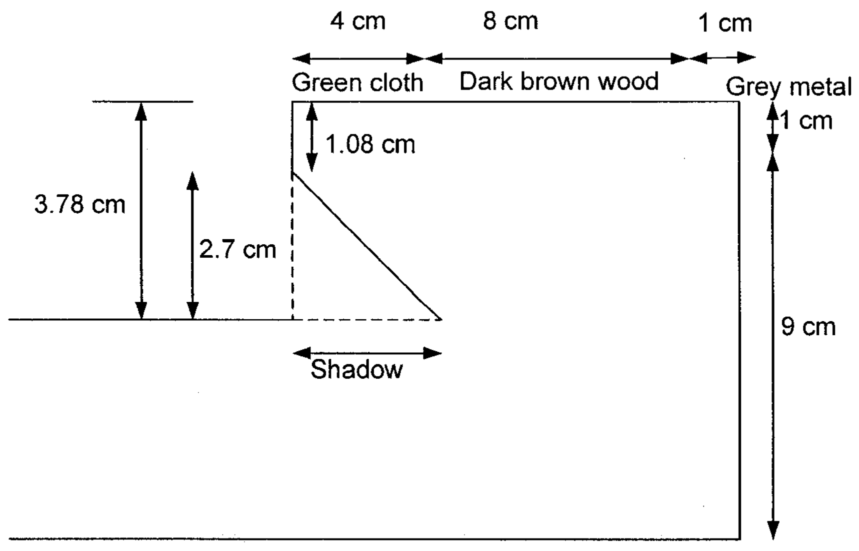


Figure 4.7-1 The place of shadow

Code example:

```
void shadowmatrix(void){
 int i;
 for(i=0;i<16;i++)
        {
          m[i]=0.0;
        }
 m[0]=m[5]=m[10]=1;
 m[7]= -1/light0_position[1];
  }
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
               glPopMatrix();
        glMatrixMode(GL_MODELVIEW);
        glPushMatrix();
```

```
glTranslatef(0,0.01,0);
            glTranslatef(light0_position[0],light0_position[1],light0_position[2]);
            glMultMatrixf(m);
            glTranslatef(-light0_position[0],-light0_position[1],-light0_position[2]);
            glColor3fv(shadow_color);
            glMaterialfv(GL_FRONT,GL_AMBIENT_AND_DIFFUSE,black);
 glMaterialfv(GL_FRONT,GL_SHININESS,dull);
 glMaterialfv(GL_FRONT,GL_SPECULAR,black);
 glBegin(GL_POLYGON);
      glVertex3f(NearRoleCoverX[5],NearRoleCoverZ[2],NearRoleCoverY[4]);
     glVertex3f(NearRoleCoverX[5],NearRoleCoverZ[2],NearRoleCoverY[5]);
     glVertex3f (NearRoleCoverX[17], NearRoleCoverZ[2], NearRoleCoverY[5]);
     glVertex3f (NearRoleCoverX[17], NearRoleCoverZ[2], NearRoleCoverY[4]);
glEnd();
            glPopMatrix();
            glPopMatrix();
 glFlush();
 glutSwapBuffers();
}
```

5 CONCLUSION

5.1 Introduction

Having discussed all the technical aspects of the project, this chapter presents the conclusions of the project. It contains what has been achieved and how this differs from what was set out to be achieved, presenting the reasons for these differences. It also contains the learning outcome. Finally, the chapter contains a section of possible future improvements. This is further work that could be carried out on the project to enhance it.

5.2 Evaluation

This section looks critically at the work carried out explaining the successes and the shortcomings of the project.

The Pros

The project set out to demonstrate the feasibility of constructing a 3D snooker table using OpenGL. I can safely say that most of the goals initially set for the snooker table were met. The result is a simulator with different viewpoints, thus achieving the project's primary goal. All the necessary components have been implemented. Features include viewing transformation, scaling, texture mapping, lighting and shading. The look of the texture mapped table cloth and the texture mapped table edge fully matches my expectations. The whole time for implementation is two months.

To this end, the project has come to a satisfactory conclusion.

The Cons

What has not been achieved is the creation of an enjoyable game. As it stands there is no ball to rolling on the table. There are suggestions as to how this may be resolved in the future improvements section. The most difficult and time-consuming aspect of the project was modeling the pockets, which are composed of many polygons. Calculating the accurate coordinate system is the most important step to model every tiny polygon.

5.3 Experiences in OpenGL

OpenGL is a fairly straight forward — although at many times confusing — concept. OpenGL gives the programmer an interface with the graphics hardware. OpenGL is a low-level, widely supported modeling and rendering software package, available on all platforms. It can be used in a range of graphics applications, such as games, CAD design, or modeling.

OpenGL is perfectly capable of producing something of this genre. OpenGL is the core graphics rendering option for many 3D games. The providing of only low-level rendering routines is fully intentional because this gives the programmer a great control and flexibility in his applications. These routines can easily be used to build high-level rendering and modeling libraries.

OpenGL is a collection of several hundred functions that provide access to all of the features that graphics hardware has to offer. Internally it acts like a state machine-a collection of states that tell OpenGL what to do. Using the API it is available to set various aspects of this state machine, including current color, blending, lighting effect, etc.

5.4 Suggestions for Future Research

This section describes elements that are beyond the scope of the project, but would be worthwhile implementing if time was available.

Add balls

As I introduced in the instruction, the snooker table is a component of playable game. Using what I have learned from this experience, I would like to add the model of balls. I would also like to explore a real time model for when the ball collides with other objects as well as particle systems for explosions, weather models and other effects.

Collision Detection

In order to get the game playable, collision detection is an essential component of snooker game. In virtual prototype, it is used to refine designs without productions of physical prototypes in the initial stage.

Sound

Virtually all games in the commercial market today will make use of music and sound effects. The game could benefit from sound effects for the engine, a sound being played when a collision occurs. Sound would increase the sense of atmosphere and reality allowing the user to become more involved in the game.

Multiplayer (and Network)

The world would be far less lonely if there are other crafts flying too. It would be possible to make the game '2-player' by opening another window and having a second set of controls (more keyboard callbacks) to control a second craft. This again expands the number of options for the game element where it would be possible to have any number of race scenarios or a destruction-derby style game. Once implemented, the natural extension for the 2 player mode is to allow it to be run over a network with users competing on different machines.

Reference

OpenGL

[1] Neider, Jackie, Tom Davis, and Mason Woo. OpenGL Programming Guide Addison-Wesley Publishing Company, 1994

[2] Angel, E. (2000). Interactive Computer Graphics, a top-down approach with OpenGL. New York: Addison-Wesley

[3] *Jeff Molofee's OpenGL Tutorials*, available at: http://nehe.gamedev.net/, provided a valuable reference for common OpenGL techniques and code samples.

[4] Peter Grogono's OpenGL Tutorials, available at: http://www.cs.concordia.ca/~grogono/gldoc.pdf,

[5] http://www.creative-computing-inc.com/opengl.htm

[6] http://www.delphi3d.net/articles/printarticle.php?article=framebuf.htm

[7] http://www.dataplus.co.jp/pdf-e/OpenGLOverview.pdf

[8] http://www.experience.epson.com.au/help/understandingcolour/COL_G/0503_3.htm

Graphics

Snooker Game on www.flyordie.com

3D live snooker game on www.etiumsoft.com

Pool Game on www.poolclub.com.