THE OPENGL FUNCTION GLULOOKAT( )

FAN WANG

A MAJOR REPORT

IN

THE DEPARTMENT

OF

COMPUTER SCIENCE

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF COMPUTER SCIENCE
CONCORDIA UNIVERSITY
MONTREAL, QUEBEC, CANADA

NOVEMBER 2002

0-612-77994-7

ABSTRACT

The OpenGL Function gluLookAt( )

Fan Wang

As any image on computer screens rendered using coloured pixels is actually two-dimensional, OpenGL applies model view transformation and projection transformation to simulate three-dimensional space in the real world. The function gluLookAt( ) is a viewing transformation specifying a viewing point, a "look at" point to which the camera or eye look at and a vector specifying upward direction.

In this major report, the author discusses the model view transformation and function gluLookAt( ) and designs two programs, the Demonstration of Function gluLookAt( ) which demonstrates a series of transformations of translations and rotations encapsulated in the gluLookAt( ) routine and the Small Town Animation System which draws a scene of a small town where there are some moving objects such as trucks and helicopter and some stationary objects such as buildings and houses, and also there is a camera that can be moved in several ways. These objects look at each other according to user's choice. The purpose of the project is to demonstrate how OpenGL function gluLookAt( ) works and the concept of model view transformation.

Acknowledgment

I would like to thank my supervisor Dr. Adam Krzyzak for his guidance and valuable advise and comments. and for his generous help on this work.

I also thank the Computer Science faculty and staff for their work of teaching and facilitating my study through years.

And my wife, for her kindness and patience.

Table of Contents

List of Tables

List of Figures

Chapter 1: Introduction

Within the past twenty years, computer graphic design has been one of the fastest developing areas in computer science and industry. Today, programmers and artists use various graphic development software tools to design industrial or scientific simulations or digital games. OpenGL is one such software tool. However, OpenGL does not provide a graphic programming environment. Instead it provides standard application program interfaces (API) for defining graphic images. Programmers must write C/C++ code to create the models from simple primitive models to sophisticated scenes. OpenGL is a software interface to graphics hardware. It is hardware independent, and can be implemented on many different platforms. Prior to OpenGL, any company developing a graphical application typically needed to rewrite the graphics part of the system for different operating system and graphics hardware. This made porting of applications very difficult and time consuming. With OpenGL, the application is created and can be reused in any system platforms and graphics hardware as long as they are compliant to OpenGL standard.

The motivation of the this report comes from the phenomenon that most new learners of OpenGL often confuse three-dimensional (3D) concepts such as coordinate system, model view transformation and projection transformation. Such difficulty becomes an obstacle of quickly getting into the amazing field of the computer graphic of OpenGL for beginners who are even skilled C/C++ programmers but in defect of geometry and 3D space knowledge. The function gluLookAt( ) as a routine of OpenGL utility library simulates the actions of “looking at” in our real lives. The author attempts uses it as a starting point and has designed and implemented the Demonstration of Function gluLookAt( ) and the small town animation system where the users can get an intuitive sense of these concepts by separating transformations within function gluLookAt( ) and by switching among different viewing points and look at points.

The next chapter (Chapter Two) reviews fundamental concepts and basic programming structure of OpenGL. Chapter Three discusses the concepts of coordinate system and model view and projection transformation. Chapter Four discusses the concept of gluLookAt( ) routine along with a formal proof of the equivalent matrix of it. Chapter Five and Fix are the design and implementation parts of the Demonstration of Function gluLookAt and the Small Town System. Chapter Seven gives the conclusion and possible problems in the project as well as the future works regarding this report.

Chapter 2: Review of OpenGL

This chapter reviews some basic concepts and program structure of OpenGL and answers the questions of what OpenGL is and what it does. Since this report mainly concerns model view transformation and gluLookAt( ) routine, other OpenGL features such as lighting, blending, texture mapping, and antialiasing are not discussed even though some of them would be applied in the implementation part of this project. Finally, limitations and constraints of OpenGL are presented.

2.1 What is OpenGL

OpenGL (Open Graphic Library) is a software interface to graphic hardware. It consists of three libraries: the Graphic Library (GL), the Graphics Library Utilities (GLU) and the Graphics Library Utilities Toolkit (GLUT). OpenGL commands with prefix gl, glu or glut correspond to these three libraries. There are about 250 distinct commands (about 200 in the core OpenGL and another 50 in the OpenGL Utility Library).

Unlike many other graphic develop software, OpenGL does not provide high level commands to describe models of three-dimensional object. Such kind of commands may used to draw much complicated shapes like houses, automobiles, aeroplanes, or part of human bodies. Using OpenGL, you build your own models consists of a series of geometric primitives. These geometric primitives are points, lines and polygons plus some modelling features provided by the OpenGL Utility Library (GLU) such as quadric surfaces, NURBS curves and surfaces. Any complex model may consist of a set of such kind of models.

OpenGL provides a set of immediately executed functions supported by its three libraries. Any drawing action is control by predefined routine. The result of such a list of actions will be desired models or scene. OpenGL support both two-dimensional (2D) and three-dimensional (3D) graphic. Many build in features make OpenGL graphic more realistic such as hidden surface removal, model view transformation, blending, anti-aliasing, texture mapping and atmospheric effects (fog, smoke etc).

Silicon Graphics initially develop the OpenGL API and lead an industry group (the OpenGL Architecture Review Board) in creating the standard. The current OpenGL API version is 1.3. OpenGL is free of licence and you can download the OpenGL library for Windows 98/NT/2K and learn much more about OpenGL at the official site of the OpenGL Architectural Review Board.

2.2 How an OpenGL Application Works

Creating images using OpenGL typically involves large amount of code, since any model is generated by a set of basic primitives including points, lines and polygons. However, OpenGL programs share common structure. Typically they have structures as follow:

- Initializing and creating window
- Drawing objects
- Handling call backs

Figure 2.1 shows a simple OpenGL program that will open a window and make the background of that window blue, then draw a red rectangle. We will describe every step of the program behaviours to illustrate the basic structure of a typical OpenGL application.

```
#include <GL\glut.h>

void display(void)
{
    glClear(GL_COLOR_BUFFER_BIT);

    // Set color
    glColor3f(1.0,0.0,0.0);

    // Draw a rectangle
    glBegin(GL_POLYGON);
      glVertex3f(-0.5, -0.5, 0.0);
      glVertex3f(0.5, -0.5, 0.0);
      glVertex3f(0.5, 0.5, 0.0);
      glVertex3f(-0.5, 0.5, 0.0);
    glEnd();

    glFlush();
}

void init(void)
{
    // Set background color
    glClearColor(0.0, 0.0, 1.0, 0.0);
}

void main (int argc, char* argv[])
{
    glutInit(&argc,argv);
    glutInitDisplayMode(GLUT_RGBA|GLUT_SINGLE);
    // Initialize and create a window
    glutInitWindowSize(500, 500);
    glutInitWindowPosition(60,40);
    glutCreateWindow("a Simple Program");
    init();
    glutDisplayFunc(display);
    glutMainLoop();
}
```

Figure 2. 1 A Simple OpenGL Program – Display a Rectangle

When the program is run, it produces a square window with width and height 500 pixels and a blue background. The #include statements in the first line load all the function headers for the GL libraries.

As any C/C++ program, the program starts running from its main function. The first few lines of the main function initialize parameters for the window. The first statement, glutInit( ) initializes the glut library.

The next function, glutInitDisplayMode( ) initializes the display. The argument here is a bit string, describing the attributes of the window. In this program, the window will be single buffered (GLUT_SINGLE) and will use the RGBA color set (GLUT_RGBA). The program will be fine if this statement is deleted since the setting here is default.

The next line, glutInitWindowSize(500, 500), requests a graphic window 500 pixels wide and high. glutInitWindowPosition(60,40), specifies the position of the window. It says that the left side of the window should be 60 pixels from left of the screen and the topside of the window should be 40 pixels from the top of the screen.

The next line, glutCreateWindow( ), creates the window and give its title "A Simple OpenGL Program". Note the window does not actually display until the function glutMainLoop( ) has been called. Function init( ) here is used to set background colour. Many other features can also be set here. Function gluMainLoop( ) takes the program into an infinite loop. The loop processes all events for the window and makes the program continually run until it is terminated.

You need write a function that displays your graphic objects. Here, the function is called display( ). You can name it in your own way. But be sure the name should be the same as the parameter name in function glutDisplayFunc( ).

The first line in display function glClear( ) is used to clear various buffers before starting to draw new objects. The colour buffer, which is cleared here, is where your graphic models actually stored.

Next statement glColor3f( ) specify the colour of your object will be drawn following this statement. All the objects in the following statement along the program execution flow will be set using this colour unless you define it again.

The following several lines are used to define the model actually displayed. Here a polygon specified by four points is defined. The codes for drawing OpenGL primitive objects is generally like this form:

```
glBegin(mode);
......
glEnd;
```

The parameter “mode” here specify weather it is a point, line or polygon.

The last statement glFlush( ), like its name, flushes the command buffer to make sure all commands have been processed before the function exits so that you can see your model displayed in the window.

2.3 Call Back Functions

OpenGL handles input events by call back functions. There are five forms of call back functions typically. They are:

- Display call back function
- Keyboard call back functions
- Mouse event call back functions
- Window reshaping function
- Idle functions

Call back function must be registered in function init( ) or main( ). The command for registering above call back functions are:

- glutDisplayFunc(display function name).
- glutKeyboardFunc(keyboard).
- glutMoseFunc(mouse_button).

- glutReshapeFunc(reshape function name).
- glutIdleFunc(idle function name).

The display call back function is where your program draws the scene. You can directly define the drawing actions here or calls drawing functions to create the mode or scene.

Keyboard call back function is called whenever a user presses a key. The function usually handles the multiple key reactions using a switch-case selection. The keyboard call back function glutKeyboardFunc(keyboard) can only recognizes keys with ASCII code and ESC, DELETE and backspace. Other keys such as the arrow keys are handle by special key call back function gluSpecialFunc(special) which is also required to be registered in initialization function using command glutSpecialFunc(special).

Similar to keyboard call back function, the mouse call back function is called whenever a user clicks a mouse key.

Window reshaping function is actually a special form of mouse call back. When a user drag the window to resize it. OpenGL calls this call back function to insure the images displayed in the window do not distorted or disappeared. The way to do this is to redefine the size of the view port.

Idle call back function is called when the program has nothing else to do. The application for idle call back function is animations. The typical realization of the idle function is to continuously moving the position of the object by increasing or decreasing its coordinates.

2.4 OpenGL Command Syntax

You may have noticed that all OpenGL functions have a prefix is either gl, glu or glut. Such a naming convention makes it clear that the statement is coming from an OpenGL library. The functions have forms like:

gl<Function_Name>( ); example: glClear( );

glu<Function_Name>( ); example: gluLookAt( ); will be discussed.

glut<Function_Name>( ); example: glutInit( );

In addition, some functions may have multiple arguments of the same type, such as vertex or color data. For such kind of functions, numbers and letters will be added between the function names and the parameters to define the number and type of arguments being passed.

Example:

```
glColor3f( 0.0, 0.0, 1.0 );
```

Statement sets current color blue. All parameter are of GLfloat type.

```
glColor4f( 0.0, 0.0, 1.0, 1.0 );
```

Statement sets current color blue with blending fact 1.0. All parameter are of type GLfloat.

```
glVertex2d( 0.5, 0.5 );
```

Statement specifies a point with both x and y coordinate is 0.5. All parameter are of type GLdouble.

```
glVertex3d( 0.5, 0.5, 0.5);
```

Statement specifies a point with x, y and z coordinate are all 0.5. All parameter are of type GLdouble.

```
glVertex4d( 0.5, 0.5, 0.5,0.0);
```

Statement specifies a point with x, y and z coordinate are all 0.5 with blending fact 1.0. All parameter are of type GLdouble.

```
GLfloat array[] = {0.0, 1.0, 0.0, 0.0};

glColor4fv(array);
```

First statement defines a 4-element array as parameter. The pointer of the array passed to glColor4fv function as parameter. The color is red and the blend fact is 0.0.

Here the letters immediately follow the Function names indicate the different argument types of these commands. The 'f' of suffix shows that the arguments are floating-point type numbers and 'b' shows double type numbers. The 'v' of suffix shows that command takes a pointer to a vector (or array) as a parameter instead of a series of individual arguments. Some OpenGL commands can accept as many as eight different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are illustrated in table 2.1.

Suffix Data Type	Corresponding C Type	OpenGL Type Definition
b 8-bit integer	Signed car	Glbyte
b 16-bit integer	Short	Glshort
i 32-bit integer	int or long	GLint, Glsizei
f 32-bit float-point	Float	GLfloat, GLclampf
d 64-bit float-point	Double	GLdouble, GLclampd
ub 8-bit unsigned integer	Unsigned char	GLubyte, GLboolean
us 16-bit unsigned integer	Unsigned short	Glushort
ui 32-bit unsigned integer	Unsigned int or unsigned long	GLuint, GLenum, Glbitfield

Table 2.1: Command Suffixes and Argument Data Types

2.5 OpenGL limitations

As we know, OpenGL does not provide a graphic development environment. Programmers cannot draw models using mouse in the graphic edit environment instead they must write a considerable large amount of C/C++ codes. This restricts the OpenGL programmers must have C or C++ programming knowledge. So OpenGL is a graphic design tool most likely for programmers not artists.

Another limitation of OpenGL is that it is not a pixel-exact specification. Therefore, OpenGL doesn't guarantee an exact match between images produced by different OpenGL implementations.

Chapter 3: Model View and Projection

This chapter discusses transformation, an important concept in OpenGL. Firstly, the understanding of the coordinate system is present. Then the descriptions of model view transformation and projection transformation are given.

3.1 Understanding Coordinate System

The first step to 3D graphic programming is to understand coordinate systems. Without enough knowledge of coordinate system, trying to program 3D graphics may become confusing.

OpenGL uses both the two-dimensional (2D) coordinate system and three-dimensional (3D) coordinate system. Two-dimensional coordinate system uses the X-axis as horizontal distance and the Y-axis as vertical distance. When there is no need to describe 3D graphics, two-dimensional cordinates is used.

It is usually easier to think of objects in three-dimension (3D) since it is close to real world. To represent three-dimension, a Z-axis must be added to reflect the distance and depth of the object from our eyes. In three-dimensional system, the Z-axis is directed into the screen and away from the eyes of the viewer. By using the Z-axis, an object can be described in three dimensions, but we all know the image on screen is actually two-dimensional rendered by horizon and vertical pixels. What we see is just art made by computer graphics.

However, OpenGL uses three-dimensional mode by default. When the program starts, the coordinate system is centered on the origin, that is, x = 0, y = 0, z = 0. The positive X-axis extends to the right direction of the window from the origin of the coordinate, the positive Y-axis extends upward from the origin to the top of the window, and the positive Z-axis points to the outside of screen toward the viewer. So we can know that the computer screen is actually the plane where z = 0.

Remember that the purpose of computer graphic programming is to create a two-dimensional image of three-dimensional objects since a flat screen has to be two-dimensional. But the programmer must think in three-dimensions. Need a programmer to decide which pixels must be drawn? No, just think about your models in a three-dimensional space with the depth inside the screen, and let the computer to calculate which pixels need to be drawn.

It seems all right up to now, nothing confusing. But the coordinate we just talk about is so-called eye coordinate system or universal coordinate system or fixed coordinate system. However, there is also another coordinate system we must to know, that is, local coordinate system or object coordinate. The concept here comes from transformation. At the beginning, if we do not apply any transformation to anything in the system. The eye coordinate system is also the local coordinate system. As we apply transformation, for instance, translate the object along the positive X-axis for a certain distance. We move the local coordinate system but eye coordinate system remains stationary. The two coordinate systems separate. We must notice that any drawing after transformation is based on local coordinate system not eye coordinate system. We will next to talk about transformation to make things more clear.

3.2 Understanding Transformation

Transformation is a computing process. In OpenGL, transformation is represented by matrix multiplication. The effect of a transformation is to move an object, or a set of objects, with respect to the universal coordinate system.

We apply transformation to:

- The model (or part of the model) – Model transformation
- The camera (or eye) – View transformation

```
The camera lens - Projection transformation
```

The OpenGL functions that manipulate transformation are as follow:

```
glMatrixMode(GL_MODELVIEW);

glLoadIdentity( );

glTranslate*(x,y,z);

glRotate*(x,y,z) ;

glScale*(x,y,z);
```

OpenGL maintains two current matrices: model view matrix and projection matrix. glMatrixMode(GL_MATRIX_MODE) define the current matrix. The state variable GL_MATRIX_MODE specifies which matrix is currently used. The default matrix is the modelview matrix. You can use the command glMatrixMode to change the value of the GL_MATRIX_MODE state variable. The possible values for the mode argument are GL_MODELVIEW, GL_PROJECTION, and GL_TEXTURE (not discussed in this report). When glMatrixMode is called, the current matrix becomes the named matrix and all subsequent transformation commands affect the current matrix.

The command glLoadIdentity( ) is used to initialize current matrix.

The command glTranslate*(x, y, z)multiplies the current matrix by a matrix that moves an object by the given *x*, *y*, and *z* values (or moves the local coordinate system by the same amounts).

The command glRotate*(x, y, z)multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counter clockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation.

The command glScale*(x, y, z) multiplies the current matrix by a matrix that stretches, shrinks, or reflects and object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. With the local coordinate system approach, the local coordinate axes are stretched by the x, y and z factors, and the associated object is stretched with them.

3.3 Model View Transformation

Model and view transformation are combined in OpenGL and defined as a single model view matrix. It is hard to understand for a new learner but just consider viewing objects in the real world. When we look at something that is moving we can imagine that it does not move at all and it is ourselves that move. Why? Since the move is relative between the viewer and the object. You may have such an experience that in the subway station, you are sitting in a train and another train is just start to departure, sometimes you may misunderstand that it is your train that moves. The same happens in the model view transformation, when a cube we are observing rotates a little bit angle clock wisely, we can think that it is our head rotates the same angle degree counter clock wisely. Of cause we need to ignore any other objects such as table or ground as long as they are not rotate the same way.

So what is modelling? In computer graphics modelling is the process of describing an object or scene so that we can eventually draw it. At the beginning, two coordinate systems, universal coordinate system and object coordinates are the same. If at this moment we draw an object, it will be located at the origin of the universal coordinate. When we apply model transformation, we change only the object coordinates. For instance, we apply a translate transformation. The universal coordinate does not change. But the object coordinates move to a particular point specified by translate command. As drawing actions happen at object coordinate origin, the object finally appear at that point instead of the origin of the universal coordinate origin.

The object and scene are created. However, we still have to know where we are looking from, what we are looking directly at (or which direction we are looking in), and which way is supposed to be the up in the final picture. The point to look from is, usually called the eye position, viewing point or camera position, the direction look into is called the view direction. The point that is to appear in the center of the picture is sometimes called the "look at point".

The function gluLookAt( ) is used to define such a view action. The call of this function is also a model view transformation. It makes the look at point (object coordinate origin) separate away from the viewing point (universal coordinate origin). These two points specify the distance of the moving. In chapter 4, we will discuss the gluLookAt( ) function in more detail.

3.4 Projection Transformation

Objects need to be in the viewing volume. The purpose of the projection transformation is to define such a viewing volume. It specifies how the object is projected on to the screen. OpenGL defines two projections, orthographic and perspective. Orthographic is set as default.

By default, viewing volume orthographic projection is the 2*2 cube where x between –1 and 1, y between –1 and 1, and z between –1 and 1, of cause, with the screen in the plane z = 0. The projection here is orthographic. It is just like you are viewing the window from an infinitely far point. You can also define the boundaries of the viewing volume. Following code segment does this job:

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glOrtho(x_left,x_right,y_bottom,y_top,z_near,z_far);
```

Where the parameter in glOrtho( ) specify the boundaries. You can compare this with the default setting as follow:

```
glOrtho(-1,1,-1,1,1,-1);
```

Any images beyond these boundaries will be invisible. Orthographic projection makes the object viewed from an infinite point. It is not the common case in the real life, that is, the farther an object is from the viewer, the smaller it appears in the final image. Rather than a rectangular box defined by orthographic projection, perspective defines viewing volume as a frustum (or truncated pyramid) with the smaller side toward viewer and the viewer's eyes is at apex of the pyramid. There are two function define the perspective projection: glFrustum( ) and glPerspective( ). The following codes describe the ways they are using.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glFrustum(x_left,x_right,y_bottom,y_top,z_near,z_far);
```

The first four parameter of glFrustum: x_left, x_right, y_bottom and y_top define the small top side of the frustum which can be thought as the screen. z_near and z_far determined the nearest and furthest points that can be viewable.

```
glMatrixMode(GL_PROJECTION);

glLoadIdentity( );

glPerspective(alpha,with/height,near,far);
```

The first parameter is an angle in degree specifies the vertical angle that the view port subtends at the eye. Its value must satisfy 0 to 180. The second parameter is the aspect ratio of the width to height of the view port. The last two parameters, as before, determine the nearest and furthest point visible.

In both projection transformations, we assume the eye is at the origin and is looking toward the negative Z orientation. Therefore, we position the model in somewhere negative Z axis.

In perspective projection, distant objects are draw smaller than near ones. This is also how a camera works. Perspective projection results in what you would probably think of as a "normal" 3D pictures. On the other hand, orthographic projection is used mainly where preserving dimensions is important, as on a mechanical drawing.

Chapter 4: gluLookAt( ) Routine

This chapter firstly discusses the concept of the function gluLookAt( ). Then the application of the function is given along with some noticeable phenomena.

4.1 Definition of gluLookAt( )

Often we need to define an arbitrary point from which a viewer can look at a scene or model around a predefined location. We have to decide where we are looking from, what we are looking directly at, in another word, which direction we are looking in, and which way is supposed to be the up direction (upward vector) in the final scene. The point to look from is, usually called the viewing point, eye position, or camera position, and the direction looked into is called the view direction. The point that is to appear in the center of the picture is usually called the look at point. The function gluLookAt( ) is used to simulate this view action. The call of this function is also a model view transformation since it changes the current model view matrix. It makes the look at point (object coordinate origin) separate away from the viewing point (universal coordinate origin) so that we can view the scene from arbitrary distance specified by viewing point and look at point. Note that gluLookAt( ) is included in OpenGL Utility Library other than the basic OpenGL Library. It essentially consists of a set of basic OpenGL commands including glTranslate*( ) and glRotate*( ). To imagine it, think about when view point and look at point is not at the same position, it is the most case, this will invokes the command glTranslate*( ) encapsulated in gluLookAt( ) routine. The result is to translate the object coordinate to look at point. If we define the upward vector with a value other than (0.0,1.0,0.0), this will invoke the command glRotate*( ) which is also encapsulated in gluLookAt( ) routine.

The function gluLookAt( ) specifies a model-view transformation that simulates looking at the model or scene from a particular viewpoint.

```
gluLookAt(eyex,eyey,eyez,
          lookatx,lookaty,lookatz,
          upx,upy,upz);
```

The function above defines a viewing matrix and multiplies it. The result viewing point is specified by eyex, eyey, and eyez. The parameters lookatx, lookaty and lookatz specify the point that the eye is looking at. So the line between the viewing point and this point becomes the line of sight. You can specify any point desired but generally it is the center of your scene in the screen. Parameters upx, upy and upz indicate upward direction.

GluLookAt( ) creates a viewing matrix derived from an eye point, a reference point indicating the center of the scene, and an UP vector.

The matrix maps the reference point to the negative Z-axis and the eye point to the origin. When a typical projection matrix is used, the center of the scene therefore maps to the center of the view port. Similarly, the direction described by the UP vector projected onto the viewing plane is mapped to the positive Y-axis so that it points upward in the view port. The UP vector must not be parallel to the line of sight from the eye point to the reference point.

Let L = (centerX-eyeX, centerY-eyeY, centerZ-eyeZ)

Let UP be the vector (upX, upY, upZ)

Then normalize as follows:

l = L / ||L||

UP' = UP / ||UP||

Finally, let r = l × UP', and u = r × l

Matrix V is then constructed as follows:

$$\begin{bmatrix} r[0] & r[1] & r[2] & 0 \\ u[0] & u[1] & u[2] & 0 \\ -l[0] & -l[1] & -l[2] & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And `gluLookAt` is equivalent to:

```
glMultMatrixf(V);

glTranslated(-eyex,-eyey,-eyez);
```

Therefore, we get the equivalent matrix as follow:

$$\begin{bmatrix} r[0] & r[1] & r[2] & -r\cdot\overline{eye} \\ u[0] & u[1] & u[2] & -u\cdot\overline{eye} \\ -l[0] & -l[1] & -l[2] & -l\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

We now prove that it can be achieved by applying a series transformation of translation and rotation. As we know function gluLookat( ) is actually a viewing transformation consists of translation and rotation transformations. The follow part of this section shows how gluLookAt( ) encapsulate those transformation of translation and rotation.

Assumption:

```
gluLookAt(eyex,eyey,eyez,
            lookatx,lookaty,lookatz,
            upx,upy,upz);
```

Eye is at origin (universal coordinate);

Eye is looking along negative Z-axis;

Eyes' up direction is aligned Y-axis;

There are two steps to transform current matrix to assumption gluLookAt( ) matrix:

1. Separates two coordinate system by rotation

2. Translate to eye position

Create a vector from eye point to lookat point:

$$\begin{bmatrix} l_x \\ l_y \\ l_z \end{bmatrix} = \begin{bmatrix} lookat_x \\ lookat_y \\ lookat_z \end{bmatrix} - \begin{bmatrix} eye_x \\ eye_y \\ eye_z \end{bmatrix}$$

Normalize the vector:

$$\hat{l} = \frac{\bar{l}}{\|\bar{l}\|}$$

$$\hat{l} = \frac{\bar{l}}{\sqrt{l_x^2 + l_y^2 + l_z^2}}$$

Map the vector to (0, 0, -1) we get lookat vector. Why -1 other than 1? Since its direction opposites with the lookat direction.

$$\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix} = V\hat{l}$$

Apply cross product to lookat vector with up vector:

$$\bar{r} = \bar{l} \times \overline{up}$$

Normalize it and make it align with (1, 0, 0):

$$\hat{r} = \frac{\bar{r}}{\|\bar{r}\|}$$

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} = V\frac{\bar{r}}{\sqrt{r_x^2 + r_y^2 + r_z^2}}$$

Again apply cross product with l to it:

$$\bar{u} = \bar{r} \times \bar{l}$$

Normalize it and make it align with (0, 1, 0):

$$\hat{u} = \frac{\bar{u}}{\|\bar{u}|}$$

$$\begin{bmatrix}0\\1\\0\end{bmatrix} = V\frac{\bar{u}}{\sqrt{u_x^2+u_y^2+u_z^2}}$$

Now we can get the result:

The universal coordinate has vector specified by (x, y, z), we need to convert eye coordinate to universal coordinate:

$$\begin{bmatrix}1&0&0\\0&1&0\\0&0&1\end{bmatrix} = V[\hat{r} \quad \hat{u} \quad -\hat{l}]$$

Since eye vector is orthogonal and normalized. We get rotation component of transformation:

$$V_{rotate} = \begin{bmatrix}\hat{r}\\\hat{u}\\-\hat{l}\end{bmatrix}$$

Finishing up the rotation component, we now do the translation component that translates it to the eye position:

$$\begin{bmatrix}\hat{r}\\\hat{u}\\-\hat{l}\end{bmatrix}\begin{bmatrix}x-eye_x\\y-eye_y\\z-eye_z\end{bmatrix} = \begin{bmatrix}x'\\y'\\z'\end{bmatrix}$$

$$\begin{bmatrix}\hat{r}\\\hat{u}\\-\hat{l}\end{bmatrix}\begin{bmatrix}x\\y\\z\end{bmatrix} - \begin{bmatrix}\hat{r}\\\hat{u}\\-\hat{l}\end{bmatrix}\begin{bmatrix}eye_x\\eye_y\\eye_z\end{bmatrix} = \begin{bmatrix}x'\\y'\\z'\end{bmatrix}$$

The result is:

$$\begin{bmatrix} & \hat{r} & & -\hat{r}\cdot\overline{eye} \\ & \hat{u} & & -\hat{u}\cdot\overline{eye} \\ & -\hat{l} & & \hat{l}\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

Comparing the above matrix with the one in the definition, we get identity matrix that completes the proof.

Now lets verify it:

First we verify lookat vector:

$$V \bullet \begin{bmatrix} \hat{l} \\ 0 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r}\cdot\overline{eye} \\ & \hat{u} & & -\hat{u}\cdot\overline{eye} \\ & -\hat{l} & & \hat{l}\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \hat{l} \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{r}\cdot\hat{l} - \hat{r}\cdot\overline{eye} \\ \hat{v}\cdot\hat{l} - \hat{u}\cdot\overline{eye} \\ -\|\hat{l}\|^2 + \hat{l}\cdot\overline{eye} \\ 0 \end{bmatrix}$$

We know: $\bar{r} = \bar{l} \times \overline{up}$, so:

$$\hat{r}\cdot\hat{l} = (\bar{l} \times \overline{up})\cdot\hat{l}$$

Since $\bar{l}$ and $\overline{up}$ are perpendicular, $\bar{l} \times \overline{up}$ is perpendicular with $\hat{l}$, therefore:

$$\hat{r}\cdot\hat{l} = (\bar{l} \times \overline{up})\cdot\hat{l} = 0$$

Similarly,

$$\hat{u}\cdot\hat{l} = (\bar{u} \times \bar{l})\cdot\hat{l} = 0$$

And we know $\|\hat{l}\| = 1$, so $-\|\hat{l}\|^2 = -1$

$\overline{eye}$ is eye vector and is perpendicular with $\hat{r}$, $\hat{u}$ and $\hat{l}$

$$\hat{r}\cdot\overline{eye} = \hat{u}\cdot\overline{eye} = \hat{l}\cdot\overline{eye} = 0$$

Therefore, we get:

$$V \bullet \begin{bmatrix} \hat{l} \\ 0 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r}\cdot\overline{eye} \\ & \hat{u} & & -\hat{u}\cdot\overline{eye} \\ & -\hat{l} & & \hat{l}\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \hat{l} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ -1 \\ 0 \end{bmatrix}$$

Which is consistent with lookat vector $\begin{bmatrix} 0 \\ 0 \\ -1 \end{bmatrix}$

Next, we verify up vector:

$$V \bullet \begin{bmatrix} \hat{up} \\ 0 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r}\cdot\overline{eye} \\ & \hat{u} & & -\hat{u}\cdot\overline{eye} \\ & -\hat{l} & & \hat{l}\cdot\overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \hat{up} \\ 0 \end{bmatrix} = \begin{bmatrix} \hat{r}\cdot\hat{up} - \hat{r}\cdot\overline{eye} \\ \hat{u}\cdot\hat{up} - \hat{u}\cdot\overline{eye} \\ -\hat{l}\cdot\hat{up} + \hat{l}\cdot\overline{eye} \\ 0 \end{bmatrix}$$

We know: $\bar{r} = \bar{l} \times \overline{up}$, so:

$$\hat{r}\cdot\hat{up} = (\bar{l} \times \overline{up})\cdot\hat{up}$$

Since $\bar{l}$ and $\overline{up}$ are perpendicular, $\bar{l} \times \overline{up}$ is perpendicular with $\hat{up}$, therefore:

$$\hat{r}\cdot\hat{up} = (\bar{l} \times \overline{up})\cdot\hat{up} = 0$$

Since $\bar{u} = \bar{r} \times \bar{l}$ and $\bar{r} = \bar{l} \times \overline{up}$, so:

$$\hat{u}\cdot\hat{up} = (\bar{r} \times \bar{l})\cdot\hat{up} = ((\bar{l} \times \overline{up}) \times \bar{l})\cdot\hat{up}$$

$\bar{l}$ and $\bar{r}$ are perpendicular, so:

$$(\bar{l} \times \overline{up}) \times \bar{l} = \hat{up}$$

And $\bar{l}$ and $\hat{up}$ are perpendicular, so:

$$\hat{u} \cdot \hat{up} = (\bar{r} \times \bar{l}) \cdot \hat{up} = ((\bar{l} \times \overline{up}) \times \bar{l}) \cdot \hat{up} = \hat{up} \cdot \hat{up} = 1$$

And we know:

$$-\hat{l} \cdot \hat{up} = 0$$

Therefore, we get:

$$V \bullet \begin{bmatrix} \hat{up} \\ 0 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r} \cdot \overline{eye} \\ & \hat{u} & & -\hat{u} \cdot \overline{eye} \\ & -\hat{l} & & \hat{l} \cdot \overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \hat{up} \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

Which is consistent with up vector $\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$

Finally, we verify the eye point:

$$V \bullet \begin{bmatrix} \overline{eye} \\ 1 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r} \cdot \overline{eye} \\ & \hat{u} & & -\hat{u} \cdot \overline{eye} \\ & -\hat{l} & & \hat{l} \cdot \overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \overline{eye} \\ 1 \end{bmatrix} = \begin{bmatrix} \hat{r} \cdot \overline{eye} - \hat{r} \cdot \overline{eye} \\ \hat{u} \cdot \overline{eye} - \hat{u} \cdot \overline{eye} \\ -\hat{l} \cdot \overline{eye} + \hat{l} \cdot \overline{eye} \\ 1 \end{bmatrix}$$

As we know that $\overline{eye}$ is eye vector and is perpendicular with $\hat{r}$, $\hat{u}$ and $\hat{l}$

$$\hat{r} \cdot \overline{eye} = \hat{u} \cdot \overline{eye} = \hat{l} \cdot \overline{eye} = 0$$

Therefore:

$$V \bullet \begin{bmatrix} \overline{eye} \\ 1 \end{bmatrix} = \begin{bmatrix} & \hat{r} & & -\hat{r} \cdot \overline{eye} \\ & \hat{u} & & -\hat{u} \cdot \overline{eye} \\ & -\hat{l} & & \hat{l} \cdot \overline{eye} \\ 0 & 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} \overline{eye} \\ 1 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Which is consistent with the origin $\begin{bmatrix}0\\0\\0\end{bmatrix}$

End of verification.

4.2 Application of gluLookAt( )

You could apply variables to any parameter to achieve animation. Look at following command:

```
gluLookAt(0.0,0.0,dist,
          0.0,0.0,0.0,
          0.0,1.0,0.0);
```

This is often used in animation. You could assign dist an initial value. Then, using idle function constantly decrease the value of variable dist, or using keyboard or mouse call back function to do it according to your instruction. Lets imagine that you are driving a car in a high way toward a toll gate. You are at position (0.0, 0.0, dist) and toll gate is at (0.0, 0.0, 0.0). The only difference between your program and car-toll gate situation is that you must stop when you arrive the gate (I hope so). In your program you can specifies the scenario in any way you want.

Lets look at what happen if command looks like this:

```
gluLookAt(0.0,0.0,10,
          10.0,0.0,0.0,
          0.0,1.0,0.0);
```

The point, which you are looking at is no longer the origin, this make sense since what is to be looked at is your freedom of choice, so is the program. You can imagine that you are in a car that is waiting before an intersection since it is still red light. You are tired of looking at the traffic light and look at to somewhere aside the road. The object you are looking for is stationary, most like a building or an advertisement board.

```
gluLookAt(0.0,0.0,10,
          x_pos,0.0,0.0,
          0.0,1.0,0.0);
```

This time, you are not looking at a building or an advertisement board. There is a car crossing the intersection in another road, which intersect with the road you drive on from the left to right. You are attracted by its color and are looking at it. An idle function is also need to handle the variable x_pos, making it increase from a negative value. Do not forget the origin now is at the center of the intersection.

What happen if both variables apply? To make things more realistic, we define the function as follow:

```
gluLookAt(0.0,0.0,dist,
          x_pos,-2.0,0.0,
          0.0,1.0,0.0);
```

Suppose you are driving a car toward an intersection bridge. Bellow the bridge there is a road intersected with the road you are driving on (that is the meaning of –2.0). Again you also attracted by a car driving on the other road. The difference is that this time you are also driving. So do not look at it so long! The variable dist and x_pos are also handled by an idle function.

In fact all the parameter can be variable, to make thing simpler, we just talk about preceding situation. In practise, eye position and the point of interested are often made to be variable and make upward vector unchanged.

4.3 Combining View Point and Perspective

The model view transformation and the projection transformation work together to make an OpenGL image program. The following form of combination is used in most situations:

```
//projection transformation
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glPerspective(alpha,with/height,near,far);
//model view transformation
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt(eyex,eyey,eyez,
          ceneterx,centery,centerz,
          upx,upy,upz);
//draw object
......
```

This seems simple, but new learners often misuse the parameters in function gluPerspective and gluLookAt so that the final image disappeared or cannot be viewed properly. The reason is that the gluLookAt routine make the object coordinate origin to be moved so that the object drawn outside the view volume or partly outside the view volume.

Lets look at following codes to understand what happen.

Code segment 1:

```
//projection transformation
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glPerspective(30, 1, 4, 8);
//model view transformation
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt(0,0,10,0,0,0,0,1,0);
//draw object cube
gluWireCube(1.0);
```

Code segment 2:

```
//projection transformation
glMatrixMode(GL_PROJECTION);
glLoadIdentity( );
glPerspective(30,1,4,10);
//model view transformation
glMatrixMode(GL_MODELVIEW);
glLoadIdentity( );
gluLookAt(0,0,10,0,0,0,0,1,0);
//draw object cube
gluWireCube(1.0);
```

In code segment 1, we specify the nearest and furthest plane of the view volume between 4 and 8 from the eye, but function gluLookAt( ) makes the center of the cube

10 away from the eye. And the cube is 1*1 in volume. 9.5 is still great than 8. The cube is located out of the view volume, so nothing is displayed.

In code segment 2, we specify the nearest and furthest plane of the view volume between 4 and 10 from the eye. Function gluLookAt( ) still makes the center of the cube 10 away from the eye. So that front half of the cube is located in view volume. Only part of the cube is displayed.

We must note that the object falls between nearest and furthest plane of the view volume does not guarantee to be view correctly. The idea here is to require the object located inside of the view volume.

Chapter 5: System Design

This chapter analyses the system requirement. Then gives the design of functionalities and models involved in the system.

5.1 The Demonstration of Function gluLookAt( )

5.1.1 Requirement Analysis

The Demonstration of Function gluLookAt( ) is a three-dimensional program which shows a universal coordinate, a object (wire cube) and a camera. It demonstrate the behaviours of function gluLookAt( ) by giving a series transformations including three rotation and one translation. These translations are encapsulated within the gluLookAt routine. The program give the users an intuit sense of the inside components of gluLookAt( ).

5.1.2 Functionality Design

When the program start running, users could control the system in the following ways:

- User is able to resize the display window by mouse drag.
- User is able to make animation by a certain key.
- User is able to quit the program by click ESC key.
- Current value and action regarding animation stages should be displayed.

- User is able to reset the program at any time.

5.1.3 Model Design

Models involved in this system defined as C/C++ functions, which includes a set of drawing actions. The following models are specified in the small town system:

- Cube: the object viewable by the camera.
- Coordinate: specifies the universal coordinate
- Camera: specifies the viewing point.

5.1.4 Animation Design

What is animation? Simply defined, animation is the illusion of movement. There are three animations triggered by key press.

- Rotate around X-coordinate.
- Rotate around Y-coordinate
- Rotate around Z-coordinate
- Move to eye position.

5.2 The Small Town Animation System

5.2.1 Requirement Analysis

The Small Town Animation System is a three-dimensional animation programm. In the town there are some buildings and houses. There is also a road across the town. In the center of the town there is a building specially called Center Building where the universal coordinate system origin is located. A helicopter is hovering above the town constantly. On the road there are two trucks running in the opposite direction each other. A camera is located at somewhere far front of the center building and is looking at the center building.

The Small Town Animation System is a demonstration of the functionality of the OpenGL Utility routine gluLookAt( ). Four viewing points are specified. There are: the center building, the helicopter, one of the truck, and the camera. The center building is a fixed viewing point. The helicopter is rotating with a predefined speed around the center building and above the town. The moving car is running along the road from left to right with a predefined speed. The camera is initially a stationary viewing point. However, using keys can change its position. So we apply four categories viewing point to the gluLookAt( ) routine:

- A fixed viewing point (center building),
- A rotated moving viewing point (helicopter),
- A directly moving viewing point (truck).

- And a moving viewing point which can be moved according to the instruction (camera).

The objects to be viewing are center building, truck and helicopter plus the road, which the truck may want to look at. So the viewing activities are as following:

- Looking at the center building from the camera
- Looking at the truck from the camera
- Looking at the helicopter from the camera
- Looking at the front road from the truck
- Looking at the center building from the truck
- Looking at the helicopter from the truck
- Looking at the center building from the helicopter
- Looking at the truck from the helicopter
- Looking at the truck from the center building
- Looking at the helicopter from the center building
- Looking at the helicopter from an arbitrary point where the distance from the helicopter and the direction toward it are fixed values.
- Looking at the helicopter from an arbitrary point where the distance from the helicopter and the direction toward it are fixed values.

In addition, to view the entire town and all models, the program provides also provides a bird view function, which makes the user view the town in three degree of the height (high, medium, low) right above the center of the town (the center building). The effect of this is mostly like the map of the town, except that there are something moving like the helicopter and trucks.

Moreover, to get a better view of the helicopter and truck, program provides nearest view from a point in front of or above the object.

5.2.2 Functionality Design

When the program start running, users could control the system in the following ways:

- User is able to resize the display window by mouse drag.
- User is able to issue the instructions by selecting commands from a menu.
- User is able to quit the program by click ESC key or by menu selection.
- User is able to make the universal coordinate be shown or hidden by menu selection.
- User is able to change the viewing point at any time by menu selection.
- User is able to set the parameters of function gluLookAt( ) by key input.
- User is able to reset the parameters as default value.
- When the system is in "Looking at the center building from the camera" mode, User is able to change the camera location by increasing or decreasing the camera height, moving camera leftward or rightward, and moving the camera forward or backward. And these could be done by key clicks. And user is able to move the camera by rotating it around center building in six orientations.

5.2.3 Model Design

Models involved in this system defined as C/C++ functions, which includes a set of drawing actions. The following models are specified in the small town system:

- Cube: a basic model, used by other models like building, truck, and helicopter, defined in header file cube.h.
- Coordinate: specifies the universal coordinate, defined in header file coordinate.h.
- Ground: specifies the ground and road acrossing the town, defined in header file ground.h.
- House: specifies the houses in the town, defined in header file house.h.
- Building: specifies the buildings in the town, defined in header file building.h.
- Helicopter: specifies the helicopter which hovering above the town, defined in header file helicopter.h.
- Truck: specifies the trucks which running on the road, defined in header file truck.h.

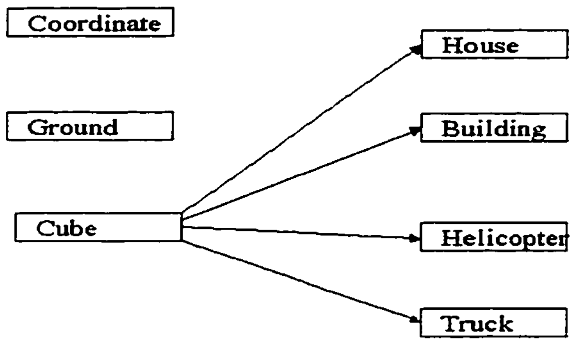

Figure 5. 1 The Model Structure of the Small Town System

5.2.4 Animation Design

There are three moving objects in the small town animation system including a helicopter and two trucks. The ideal of designing such an animation is to show that the function gluLookAt( ) is not constrained to stationary objects. Both view point and look at point can be continuously moved.

The "looking at" actions in this system mostly involves moving objects. We can even look at a moving truck from a flying helicopter and vice versa. Some time these three moving objects are all moving in the viewing volume. The result is that we can see all of them. Note that the gluLookAt( ) function define only look at point not the object we can see. All objects in the viewing volume will be displayed at the screen no matter whether or not we are looking at them.

Chapter 6: System Implementation

This chapter describes the main structures of the Demonstration of Function gluLookAt( ) and the Small Town Animation Systems, including the functions and models involved.

6.1 The Demonstration of Function gluLookAt( )

This program has only one file: lookat.cpp. The animation stages is controlled in the display( ) function by a parameter move_seq with initial value 0 which makes the system keep stationary. When the user press down key, its value increment by 1, the program enters the first animation stage. When the user press down key again, the value increment again and the program enters the second animation stage and so on. When it finish all three animation stage, the value of move_seq reset to 0. at any time user can press 'r' key to reset the program to initial state. Functions of the program are described as follow:

```
void draw_camera( );
```

This function defines the camera which denotes the viewing point.

```
void draw_up( );
```

This function defines the up vector of the camera.

```
void draw_coordinate( );
```

This function defines the universal coordinate system.

```
void reset( );
```

This function reset the parameter including camera position, rotation angle and the parameter that define the animation stage.

```
void showstring(GLfloat x,GLfloat y,char *string);
```

This function is used to display string of information.

```
void display( );
```

This is the display function that controls the animation by a switch-case structure, draw camera, cube, coordinate, up vector, lookat vector and display information.

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

This function defines keyboard call back activities including quitting from the program and resetting the position of camera and cube. The function is registered at function init( ).

```
void special(int key,int x, int y);
```

This function defines special call back activity, which triggers the animation stages. The function is registered at function init( ).

```
void reshape(int width,int height);
```

This function handles the change of the window made by user's mouse drag of the window such that the scene in the window does not distort. The function is registered at function init( ).

```
void init(void);
```

This function registers idle and call back functions including: display( ), keyboard( ), and special( ).

```
void main(int argc,char* argv[]);
```

This function is C++ main function. The function creates window for display. Call init( ) for initialization and call glutMainLoop( ) to process events, making the program running until it is terminated.

6.2 The Small Town System

6.2.1 The header Files

6.2.1.1 Coordinate.h

This file has only one function:

```
void show_coordinate( );
```

The function display positive X-axis, Y-axis and Z-axis with different color.

6.2.1.2 Ground.h

This file defines the ground and the road across the town. There are three functions in this header file:

```
void quadg(int v0,int v1,int v2,int v3);
```

This function defines the ground with. The boundaries of the ground are specified by the parameters.

```
void quadr(int v0,int v1,int v2,int v3);
```

This function defines the road across the town. The location, length and the width of the ground are specified by the parameters.

```
void quadl(int v0,int v1,int v2,int v3);
```

This function defines the line on the road. The location, length and the width of the ground are specified by the parameters. It is called line but actually is a narrow quadrangle.

```
void ground( );
```

This function calls quadg( ), quadr( ) and quadl( ) to draw the ground, road and the line on the road. And specifies their colors respectively.

6.2.1.3 Cube.h

This file provides three types of cubes, cube1 and cube2. Cube1 is used by model building. Cube2 is used by model house, helicopter and truck. There are three functions in this header file:

```
void face(int v0,int v1,int v2,int v3);
```

The function defines a face based on four points provided. The function is used by function cube1 and cube2.

```
void cube1(GLfloat color1[],GLfloat color2[]);
```

The function defines an incomplete cube, which consists of right and left side with color specified by colo1 and top side with color specified by color2. The function is used by model building.

```
void cube2( );
```

The function defines a simple cube with six faces. The function is used by model house, helicopter and truck.

6.2.1.4 House.h

This file defines two types of houses with deferent colors and orientation. They are house1 and house2. The figure 6.1 gives an example of the house. There are two functions in this header file:

```
void house1(GLfloat x_pos,GLfloat z_pos);
```

This function draws house 1. The function draws its roof and calls function cube2( ) to draw the body.

```
void house2(GLfloat x_pos,GLfloat z_pos);
```

This function draws house 2. It is similar to previous function but has different colors and orientation of the house.

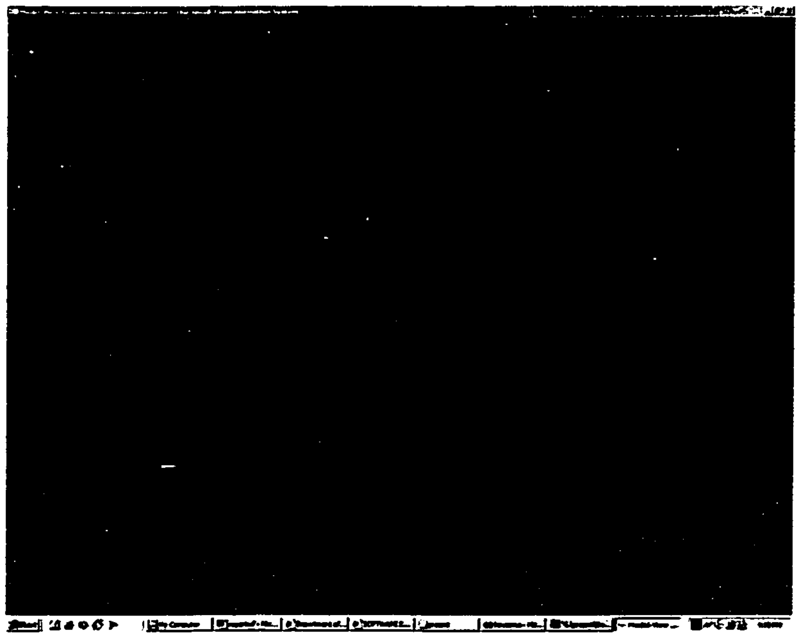

Figure 6. 1 an Example of a House

6.2.1.5 Building.h

This file defines three types of building with deferent shapes and colors. They are building1, building2 and center building which located at the center of the town where the universal coordinate is display according to users' instruction. The figure 6.2 gives an example of the building. There are three functions in this header file:

```
void read_picture( );
```

This function reads image file as texture to build the building wall.

```
void texture_wall( );
```

This function uses texture to build the front and back wall of the building.

```
void center_building( );
```

This function draws center building. The function calls function cube1( ) to build the building body and call texture_wall( ) to building the front and back wall of the building. The center building is located at the universal coordinate origin.

```
void building1(GLfloat x_pos, GLfloat z_pos);
```

This function draws building 1. The function calls function cube1( ) to build the building body and call texture_wall( ) to building the front and back wall of the building. The building's location is specified by x_pos and z_pos. The y coordinate is predefine within the function by transformation so that the building is sitting on the ground.

```
void building2(GLfloat x_pos, GLfloat z_pos);
```

This function draws building 2. The function calls function cube2( ) to build the building body and call texture_wall( ) to building the front and back wall of the building. The building's location is specified by x_pos and z_pos. The y coordinate is predefine within the function by transformation so that the building is sitting on the ground.

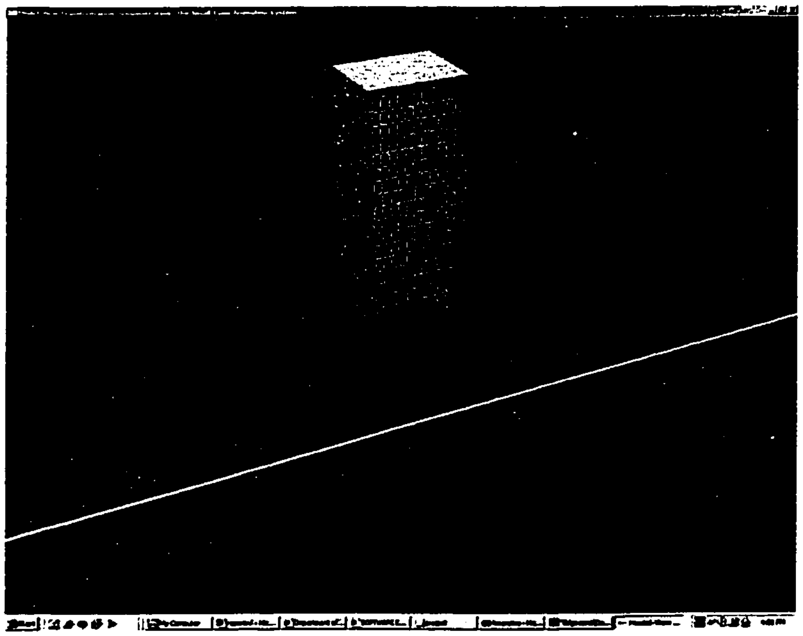

Figure 6. 2 an Example of a Building

6.2.1.6 Helicopter.h

This file has only one function:

```
void helicopter( );
```

It has a series of drawing activities that specify different part of the helicoipter including the body, wings, link between the body and wings, the foot, the links between the foot and body, the tail and the link between the tail and body. And lighting is defined to make it more realistic.

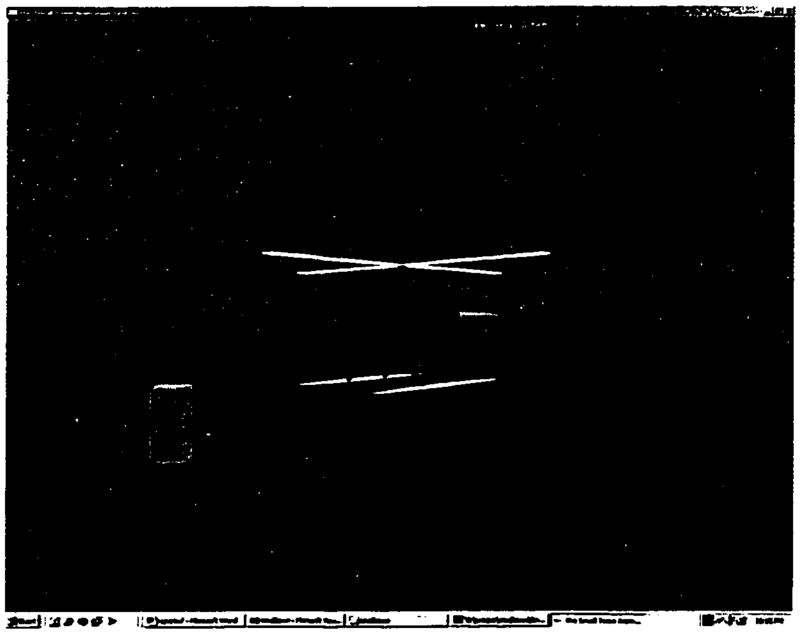
Figure 6. 3 an Example of a Helicopter

6.2.1.7 Truck.h

This file defines two trucks. The first truck is running from the left direction to the right by viewer observing. Another truck is running in opposite direction. The first car is also defined as a viewing point. The model truck is made up of two scaled cubes as the body and four cylinder components (consist of one cylinder and four disks respectively) as its wheels. There are three functions in this header file:

```
void wheel( );
```

This function define the model wheel used by model truck.

```
void truck( );
```

This function is the specification of the model truck. It defines the shapes and colors of the truck. The function is called by function truck1 and truck2.

```
void truck1( );
```

This function calls function truck( ) to make the model and define the initial position and driving direction of the truck1.

```
void truck2( );
```

This function calls function truck( ) to make the model and define the initial position and driving direction of the truck2.

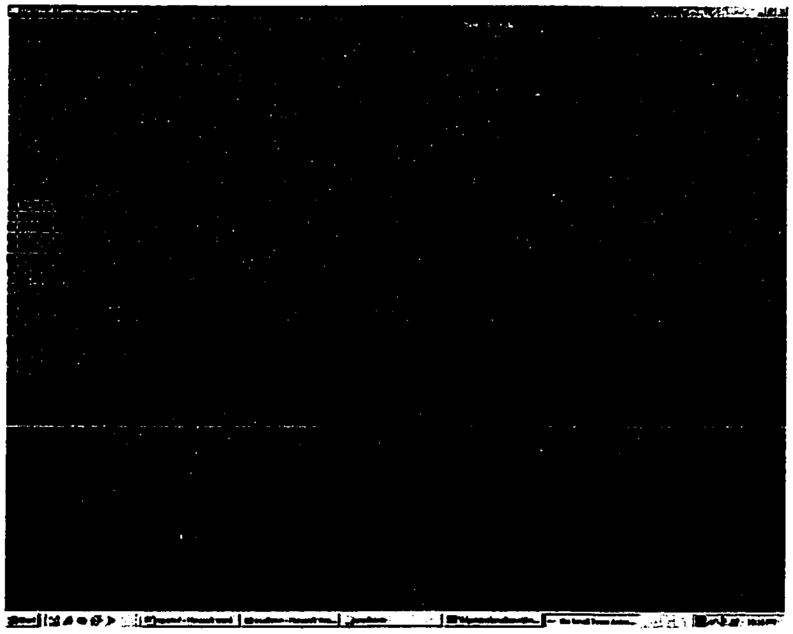

Figure 6. 4 an Example of a Truck

6.2.2 The Main File SmallTown.cpp

6.2.2.1 Function Description

This is the main file of the small town animation system. Its functionality includes:

- Defines and creates the window.
- Makes the window resizable.
- Specifies the image features such as display mode and antialiasing.
- Draws the stationary small town with a road, some buildings and houses.
- Draws one helicopter and two trucks.
- Defines the idle functions to make helicopter flying and trucks running.
- Defines viewing volume and viewing points.
- Defines menu and submenus.
- Defines keys and mouse call back functions.

The functions in this file and their functionality are described as follow:

```
void input_para( );
```

This function allows the user input parameters from the keyboard for function gluLookAt( ). The function is called by function initialize_menu( ).

```
void reset( );
```

This function reset the parameters of gluLookAt( ) and position of the helicopter and trucks. The function is called by function initialize_menu( ).

```
void set_menu(int set);
```

This function defines menu selection activities. The function is called by function initialize_menu( ).

```
void showstring(GLfloat x, GLfloat y, char *string);
```

This function displays the current view model state of the program at the top of the display window.

```
void choose_view(int select);
```

This function defines the viewing point selection activities. The function is called by function initialize_menu( ).

```
void choose_height(int select);
```

This function defines the height of the bird view to the town. The function is called by function initialize_menu( ).

```
void choose_direction(int select);
```

This function specifies the direction to which the helicopter or truck is looked from a relative fixed and not far away point. This function is called by function choose_direction_heli(int select) and void choose_direction_truck(int select).

```
void choose_direction_heli(int select);
```

This function defines a best view point to watch a helicopter from up or front orientation. The function calls choose_direction(int select) to specifies the orientation.

```
void choose_direction_truck(int select);
```

This function defines a best view point to watch a truck from up or front orientation. The function calls choose_direction(int select) to specifies the orientation.

```
void initialize_menu(void);
```

This function defines the main menu, menu and third menu. The function is called by function init( ).

```
void fly_run(void);
```

This is the idle function used to define the animation features of the helicopter and trucks. The function is registered at function init( ).

```
void town( );
```

This function draws grounds (includes road), houses, buildings and coordinate (if necessary). The function is called by function display( ).

```
void display( );
```

This function is the display function. it calls other model specification functions to draw the entire scene. The display sequence is as follow:

- Model view specification (17 selections, described in section 6.2.2).
- Draw town.
- Draw helicopter.
- Draw truck1.
- Draw truck2.

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

This function defines keyboard call back activities including quitting from the program and moving (moving in direct line) viewing point when the program is in "camera looks at center building" mode. The function is registered at function init( ).

```
void special(int key,int x,int y);
```

This function defines special call back activity which moving (rotating around X, Y an Z axis) viewing point when the program is in "camera looks at center building" mode. The function is registered at function init( ).

```
void reshape(int width,int height);
```

This function handles the change of the window made by user's mouse drag of the window such that the scene in the window does not distort. The function is registered at function init( ).

```
void init(void);
```

This function registers idle and call back functions including: fly_run( ), display( ), keyboard( ), and special( ). The function also used to do some initialization like initializing menu by calling initialize_menu( ) and initializing texture image by calling read_picture( ) and initializing antialiasing features.

```
void main(int argc,char* argv[]);
```

This function is C++ main function. The function creates window for display. Call init( ) for initialization and call glutMainLoop( ) to process events, making the program run until it is terminated.

6.2.2.2 Model View Specification

The main purpose of this report is to illustrate the concept of gluLookAt routine and demonstrate its functionality. To do this, the small town system applies up to 11 model view specifications which has 4 categories:

- Stationary viewing point to stationary model
- Stationary viewing point to moving model
- Moving viewing point to stationary model
- Moving viewing point to moving model

In addition, the moving viewing point also provides 2 forms of moving:

- Moving automatically powered by idle function
- Moving according to instruction powered by keyboard call back functions

The user can click mouse right key to invoke a menu, by selecting "Select view pint model" a sub menu appear so that the user can choice the mode view selections. This can be done whenever the program is running to switch among those selections so that the user gets an intuitionistic feel about the behavior of the gluLookAt( ) function. The eleven model view specifications are describes as follow:

1. Looking at the Center Building from the Camera:

```
gluLookAt(eyex,eyey,eyez,
          0.0,0.0,0.0,
          0.0,1.0,0.0);
```

Eye position is the camera position and model position is the center building position (universal origin). Variables apply to camera position. Viewing point can be moved by increasing or decreasing X, Y or Z coordinates of the camera position and handled by call back function keyboard( ). Viewing point moving follows a straight line. Viewing point also can be moved by rotating around X, Y or Z axis and handled by call back function special( ). Viewing point moving follows an arc line. Center building position keeps stationary.

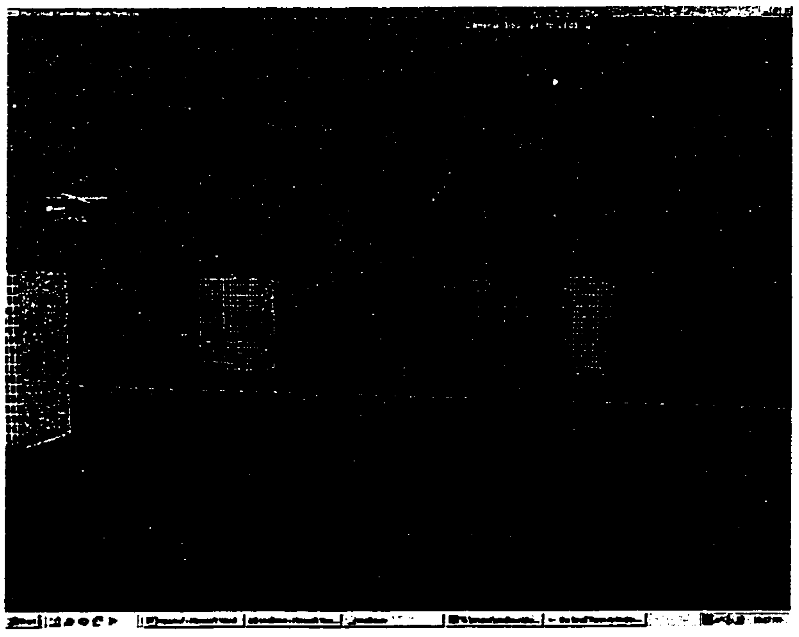

Figure 6. 5 Looking at the Center Building from the Camera

2. Looking at the truck from the Camera

```
gluLookAt(eyex,eyey,eyez,
          truck1_x,-1.0,3.5,
          0.0,1.0,0.0);
```

Eye position is the camera position and model position is the truck1 position. Variables apply to camera position and truck position. Viewing point can be moved by increasing or decreasing X, Y or Z coordinates of the camera position and handled by call back function keyboard( ). Viewing point moving follows a straight line. Since truck move in straight line only X coordinate increases and is handled by idle function fly_run( ). Truck moving follows a straight line.

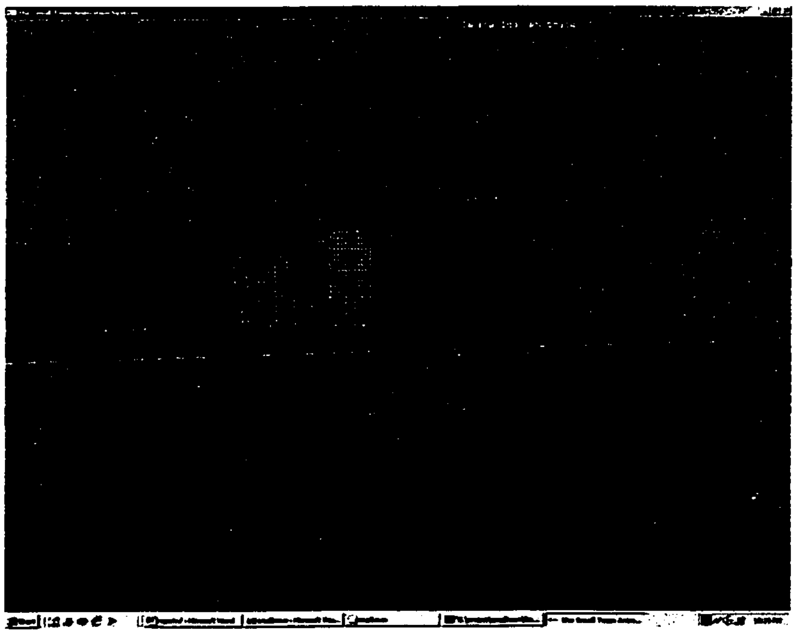

Figure 6. 6 Looking at the Truck from the Camera

3. Looking at the helicopter from the Camera

```
gluLookAt(eyex,eyey,eyez,
          heli_x,4.5,heli_z,
          0.0,1.0,0.0);
```

Eye position is the camera position and model position is the helicopter position. Variables apply to camera position and helicopter position. Viewing point can be moved by increasing or decreasing X, Y or Z coordinates of the camera position and handled by call back function keyboard( ). Viewing point moving follows a straight line. Helicopter rotates around Y coordinate in a fixed height (y = 4.5), only X and Z coordinate change. Rotation radium specified by initial value of heli_x and heli_z. Rotation is handled by idle function fly_run( ).

Figure 6. 7 Looking at the helicopter from the Camera

4. Looking at the Front Road from the Truck

```
gluLookAt(truck1_x,-1.0,3.5,
          truck1_x+10,-1.0,3.5,
          0.0,1.0,0.0);
```

Eye position is the truck1 position and model position is the point truck1 position with X coordinate plus10 . Variables apply to truck1 position. Viewing point is moved by increasing truck1_x and handled by idle function keyboard( ). Viewing point moving follows a straight line. The position of the point been viewed moves the same way as the truck.

Figure 6. 8 Looking at the Front Road from the Truck

5. Looking at the Center Building from the Truck

```
gluLookAt(truck1_x,-1.0,3.5,
             0.0,0.0,0.0,
             0.0,1.0,0.0);
```

Eye position is the truck1 position and model position is the center building position (universal origin). Variables apply to truck1 position. Viewing point is moved by increasing truck1_x and handled by idle function keyboard( ). Viewing point moving follows a straight line. Center building position keeps stationary.

Figure 6. 9 Looking at the Center Building from the Truck

6. Looking at the Helicopter from the Truck

```
gluLookAt(truck1_x,-1.0,3.5,
          heli_x,4.5,heli_z,
          0.0,1.0,0.0);
```

Eye position is the truck1 position and model position is the helicopter position. Variables apply to truck1 position and helicopter position. Viewing point is moved by increasing truck1_x and handled by idle function keyboard( ). Viewing point moving follows a straight line. Helicopter rotates around Y coordinate in a fixed height (y = 4.5), only X and Z coordinate change. Rotation radium specified by initial value of heli_x and heli_z. Rotation is handled by idle fnction fly_run( ).

Figure 6. 10 Looking at the Helicopter from the Truck

7. Looking at the Truck from the Helicopter

```
gluLookAt(heli_x,4.5,heli_z,
          truck1_x,-1.0,3.5,
          0.0,1.0,0.0);
```

Eye position is the helicopter position and model position is the truck1 position. Variables apply to helicopter position and truck1 position. Viewing point is the helicopter postion. Helicopter rotates around Y coordinate in a fixed height (y = 4.5), only X and Z coordinate change. Rotation radium specified by initial value of heli_x and heli_z. Rotation is handled by idle fnction fly_run( ). Since truck move in straight line only X coordinate increases and is handled by idle function fly_run( ). Truck moving follows a straight line.

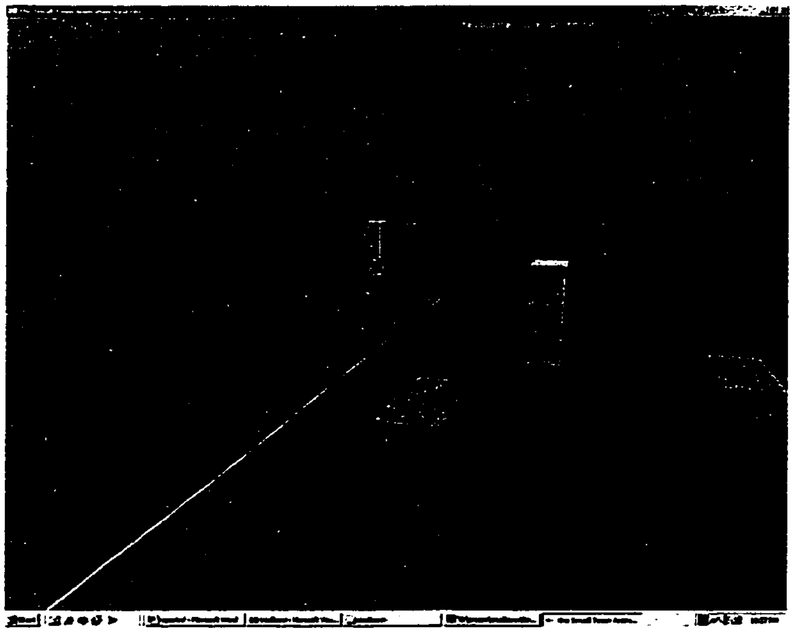

Figure 6. 11 Looking at the Truck from the Helicopter

8. Looking at Center Building from the Helicopter

```
gluLookAt(heli_x,4.5,heli_z,
          0.0,0.0,0.0,
          0.0,1.0,0.0);
```

Eye position is the helicopter position and model position is the center building position (universal origin). Variables apply to helicopter position. Viewing point is the helicopter postion. Helicopter rotates around Y coordinate in a fixed height (y = 4.5), only X and Z coordinate change. Rotation radium specified by initial value of heli_x and heli_z. Rotation is handled by idle fnction fly_run( ). Center building position keeps stationary.

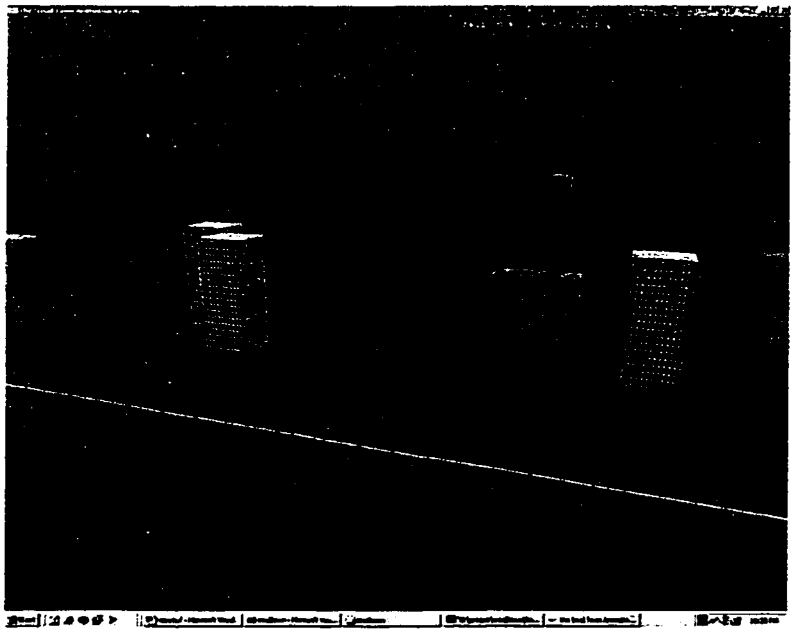

Figure 6. 12 Looking at Center Building from the Helicopter

9. Looking at Helicopter from the Center Building

```
gluLookAt(0.0,2.0,1.0,
          heli_x,4.5,heli_z,
          0.0,1.0,0.0);
```

Eye position is the center building position (universal origin) and model position is the helicopter position. Variables apply to helicopter position. Viewing point is center building position and is stationary. Helicopter rotates around Y coordinate in a fixed height (y = 4.5), only X and Z coordinate change. Rotation radium specified by initial value of heli_x and heli_z. Rotation is handled by idle function fly_run( ).

Figure 6. 13 Looking at Helicopter from the Center Building

10. Looking at the Truck from the Center Building

```
gluLookAt(0.0,2.0,1.0,
          truck1_x,-1.0,3.5,
          0.0,1.0,0.0);
```

Eye position is the center building position (universal origin) and model position is the truck position. Variables apply to helicopter position. Viewing point is center building position and is stationary. Since truck move in straight line only X coordinate increases and is handled by idle function fly_run( ). Truck moving follows a straight line.

Figure 6. 14 Looking at the Truck from the Center Building

11. Bird View of the Town (Three Levels: High, Middle and Low)

```
gluLookAt(0.0,bird_height,0.0,
          0.0,0.0,0.0,
          0.0,1.0,0.0);
```

Eye position is the point with height specified by bird_height and above the center building position (universal position) and model position is the center building position. Both eye and model position is stationary. The value of enum variable is specified by constants HIGH, MIDDLE and LOW which can be selected by the user through the menu and is handled by function choose_height( ).

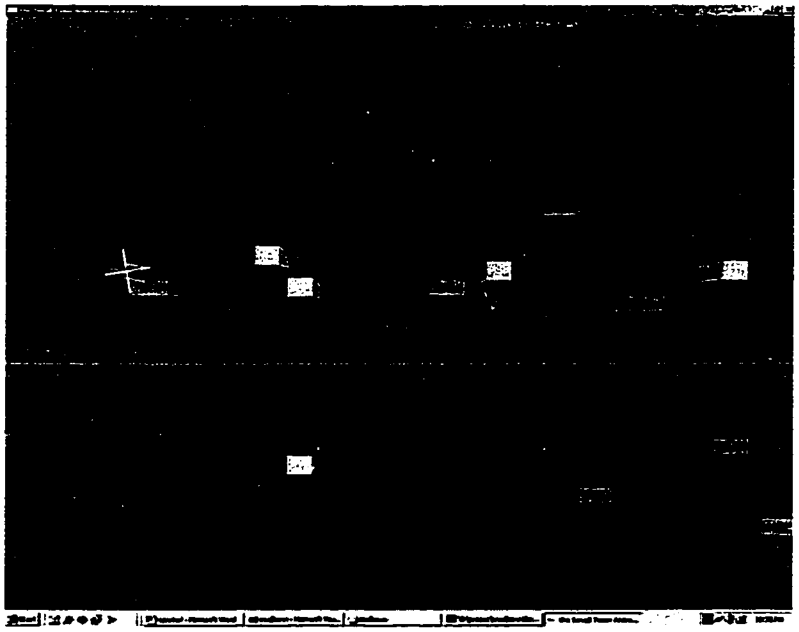

Figure 6. 15 Bird View of the Town (Level: Low)

12. See Helicopter (Two Orientations: Horizon, Vertica)

```
gluLookAt(heli_x,4.5+see_dist_y,heli_z+see_dist_z,
          heli_x,4.5,heli_z,
          0.0,up_y,up_z);
```

Sometimes, User may be interested in viewing an object suck as the helicopter or the truck in detail from a not far away viewing point. This and next function does this job. Eye position in this function have two choices: one is above the helicopter and the distance is specified by see_dist_y. Another one is in front of the helicopter with the distance specified by see_dist_z. However one of see_dist_y and see_dist_z would be zero. Weather the orientation is horizontal or vertical is specified by the value of upward vector up_y and up_z. Also, one of up_y and up_z would be zero. All these is handled by function choose_direction_heli( ) according users' instruction through menu selection.

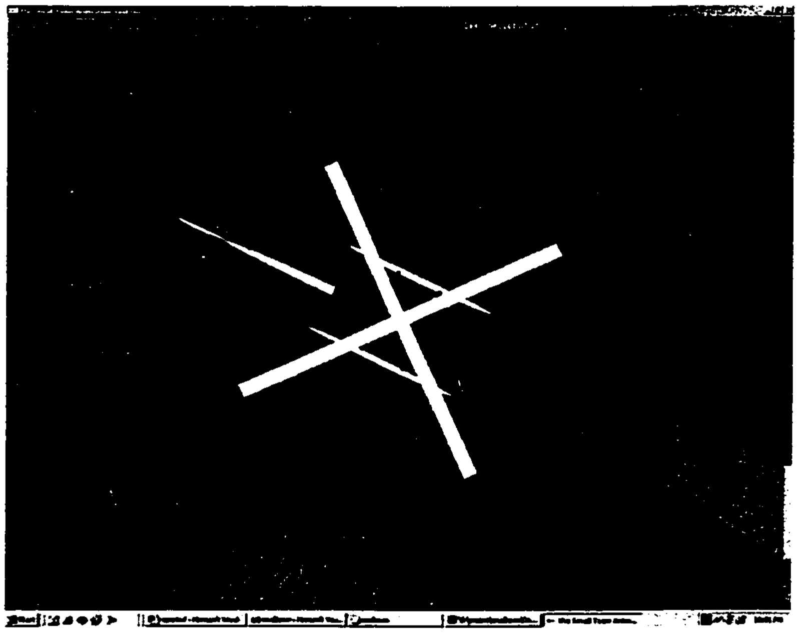

Figure 6. 16 See Helicopter (Orientation: Vertical)

13. See Truck (Two Orientations: Horizon, Vertica)

```
gluLookAt(truck1_x,see_dist_y-1.0,see_dist_z+3.5,
          truck1_x,-1.0,3.5,
          0.0,up_y,up_z);
```

This function is similar with previous one. However, the looking at object is the truck other than the helicopter.

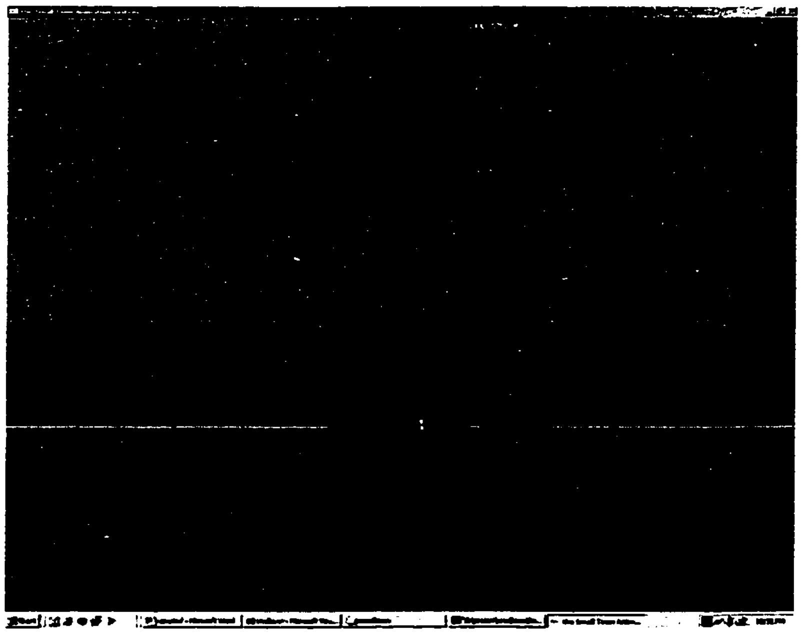

Figure 6. 17 See Truck (Orientation: Horizon)

Chapter 7: Conclusion and Future Work

The report shows that to grasp the concept and application of function gluLookAt( ) is a good start point of getting to know graphic design in OpenGL. "Two points and one direction" makes it share many common and similar points as in the real world. The function gluLookAt( ) itself is a good illustration of model view transformation. As a result, beginners to OpenGL could much more comfortable with those 3D concepts such as coordinate and transformation.

How about projection transformation? Without projection transformation, the model view transformation will be useless. In the section 4.3, I also talk about the problem raised from the combination of these two transformations and how to avoid them. However, life is not easy, it seems to be a tricky part of OpenGL in practice. If you run my implementation of "a Small Town Animation System", when two trucks pass by each other, there are some parts of the truck in opposite direction are missing. The reason is those parts are running beyond our view volume, I adjusted the parameter in perspective projection function glPerspective( ). It may be improved but never removed. Is there any better way to solve the problem to make the defects small enough so that we can't feel it? I hope so.

References

[1] Mason Woo, Jackie Neider, and Tom Davis, *OpenGL: Programming Guide, Second Edition, The Official Guide to Learning OpenGL, Version 1.1*, Addison-Wesley Developers Press, 1996

[2] Peter Grogono, *Getting Started with OpenGL, Course Note for COMP 471 and COMP 676*, Department of Computer Science, Concordia University, 1998

[3] Mike Morrison, *the Magic of Computer Graphics, First Edition*, Sams Publishing, Indianapolis Indiana, 1995

[4] Olin Lathrop, *the Way Computer Graphics Works, First Edition*, John Wiley & Sons, Inc, 1997

[5] Edward Angel, *Interactive computer graphics: a top-down approach with OpenGL*, Addison Wesley Higher Education, 2001

[6] Francis S. Hill, Jr, *the Computer Graphics Using OpenGL, second Edition*, Prentice Hall, A Division of Pearson Education, Upper Saddle River, 2000

[7] http://www.cs.arizona.edu/classes/cs433/spring02/opengl/

[8] http://www.eecs.tulane.edu/www/Terry/OpenGL/

[9] http://www.css.tayloru.edu/~btoll/resources/graphics/opengl/gltulane/

[10] http://www.opengl.org/

Appendix: User's Guide

1. System Requirements

The small town system is developed using the Microsoft Visual C++ version 6.0 and OpenGL libraries in Microsoft Windows98/2000 operating system platform. OpenGL system library files for gl glu and glut library are opengl32.lib, glu32.lib and glut32.lib respectively. Since opengl32.lib and glu library glu32.lib come with windows, so you don't need to set up them. The only one you need to set up is glut library.

First, you need to download its zip file from Microsoft homepage and unzip it, then put glut32.dll at location c:\winnt\system32, put glut32.lib at location c:\Program

Files\Microsoft Visual Studio\VC98\Lib and put glut.h at location c:\Program Files\Microsoft Visual Studio\VC98\Include\GL. After you compile the program, you need to add glut32.lib, opengl32.lib and glu32.lib to the linker.

2. Operation Guide

2.1 the Demonstration of Function gluLookAt( )

There are three keyboard operations:

- Down arrow – Trigger the animation stages.
- R(r) – Reset the program to the initial state.

- ESC – Exit program.

2.2 the Small Town Animation System

2.2.1 Keyboard operations

- F (f) – Move the camera forward by decreasing its Z coordinate.
- B (b) – Move the camera backward by increasing its Z coordinate.
- U (u) – Move the camera upward by increasing its Y coordinate.
- D (d) – Move the camera downward by decreasing its Y coordinate.
- R (r) – Move the camera rightward by increasing its X coordinate.
- L (r) – Move the camera leftward by decreasing its X coordinate.
- Left arrow – Move the viewpoints by rotating it around vertical axis of the center building clock wisely.
- Right arrow – Move the viewpoints by rotating it around vertical axis of the center building counter clock wisely.
- Up arrow – Move the viewpoints by rotating it around left-rightward horizontal axis of the center building clock wisely.
- Down arrow – Move the viewpoints by rotating it around left-rightward horizontal axis of the center building counter clock wisely.
- Page Up – Move the viewpoints by rotating it around front-backward horizontal axis of the center building clock wisely.
- Page Down – Move the viewpoints by rotating it around front-backward horizontal axis of the center building counter clock wisely.

- ESC – Exit program.

2.2.2 Menu Items:

Main menu:

- Show/Hide Coordinate – show (when it is hidden) or hide (when it is showing) the universal coordinate
- Select View Point – Open the Sub menu to select view points
- Enter the parameters from keyboard – Allow the user input parameters for gluLookAt( ) function from keyboard
- Reset – reset the parameters of the gluLookAt function
- Quit – Terminate the program

Sub menu:

- Bird View of Town – Open the third menu to select bird view of the town by levels.
- See Helicopter – Open the third menu to select the orientation for seeing the helicopter
- See Truck – Open the third menu to select the orientation for seeing the truck.
- Camera to Center Building – Make the camera looking at the center building.
- Camera to Truck – Make the camera looking at the truck.
- Camera to Helicopter – Make the camera looking at the helicopter.
- Truck to Road – Looks at the front road from the running truck.
- Truck to Center Building – Looks at the center building from the running truck.

- Truck to Helicopter – Looks at the fly helicopter from the running truck.
- Helicopter to Truck – Looks at the running truck from the flying helicopter.
- Helicopter to Center Building – Looks at the center building from flying helicopter.
- Center Building to Helicopter – Looks at the flying helicopter from the center building.
- Center Building to Truck – Looks at running truck from the center building.

Third menu:

- High – Set the height of the view point to be the greatest value.
- Middle – Set the height of the view point to be the value between greatest value and smallest value.
- Low – Set the height of the view point to be smallest value.
- Horizon – Set the orientation of seeing the helicopter or truck be horizontal
- Vertical – Set the orientation of seeing the helicopter or truck be vertical