Creating realistic images for Visual Analysis based on OpenGL
by Daniel Yong Yue

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science
Montana State University

Abstract:
The creation of realistic images is an important aspect in the field of architectural, design. A fundamental difficulty in achieving total visual realism is the complexity of the real world. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects.

Visual Analysis is a software package which is used by architects in the building design, but it lacks the features of rendering realistic images. This thesis discussed the various techniques for rendering realistic images, including coloring, lighting, shading, material properties, multiple lights, viewing and other techniques.

In this thesis we describe 3D realistic images that have been rendered with enhanced visual comprehensibility that are satisfactory for building design. Empirical methods based on mathematical models and results have been presented and the performance of Visual Analysis has been improved.

Creating Realistic Images for Visual Analysis Based on OpenGL

by

Daniel Yong Yue

A thesis submitted in partial fulfillment
of the requirements for the degree

of

Master of Science

in

Computer Science

Montana State University
Bozeman, Montana

July 1997

APPROVAL

of a thesis submitted by

Daniel Yong Yue

This thesis has been read by each member of the thesis committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the College of Graduate Studies.

July 2nd 1997
Date

J Dwight Stanley
Chairperson, Graduate Committee

Approved for the Major Department

July 2nd 1997
Date

J Dwight Stanley
Head, Major Department

Approved for the College of Graduate Studies

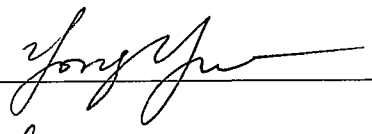
7/9/97
Date

Graduate Dean

STATEMENT OF PERMISSION TO USE

In presenting this thesis in partial fulfillment of the requirements for a master's degree at Montana State University, I agree that the Library shall make it available to borrowers under rules of the Library.

If I have indicated my intention to copyright this thesis by including a copyright notice page, copying is allowable only for scholarly purposes, consistent with "fair use" as prescribed in the U.S. Copyright Law. Requests for permission for extended quotation from or reproduction of this thesis in whole or in parts may be granted only by the copyright holder.

Signature 

Date July 2, 1997

ACKNOWLEDGMENTS

I would like to thank my thesis advisor, Dr. J. Denbigh Starkey, for his advice and support while supervising this thesis. I would also like to thank Professor Ray Babcock and Barbara Ellestad for their kind guidance and serving on my committee.

I am especially grateful to Professor R. D. VanLuchene of Civil Engineering Department for his many helpful comments and helping me finish this project and the thesis.

Finally, I deeply appreciate the patience, understanding, and love provided by my wife, Xiao Li; She makes it all worthwhile.

Contents

List of Tables

List of Figures

Abstract

The creation of realistic images is an important aspect in the field of architectural design. A fundamental difficulty in achieving total visual realism is the complexity of the real world. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects.

Visual Analysis is a software package which is used by architects in the building design, but it lacks the features of rendering realistic images. This thesis discussed the various techniques for rendering realistic images, including coloring, lighting, shading, material properties, multiple lights, viewing and other techniques.

In this thesis we describe 3D realistic images that have been rendered with enhanced visual comprehensibility that are satisfactory for building design. Empirical methods based on mathematical models and results have been presented and the performance of Visual Analysis has been improved.

Chapter 1

Introduction

1.1 Visual Realism

There are many different approaches for rendering images, for example, creating 2D and 3D graphs of mathematical, physical, and economic functions; histograms, bar and pie charts; task-scheduling charts; inventory and production charts; multimedia systems; simulation and animation for scientific visualization and entertainment; computer-aided drafting and design. Also, in computer-aided design (CAD), users can employ interactive graphics to design components and systems of mechanical, electrical, electromechanical, and electronic devices, including structures such as automobile bodies, airplane and ship hulls, computer networks and building. In this thesis, we take the approach of displaying realistic simulation results for building design.

In what sense a picture can be said to be *realistic* is a subject of much scholarly debate [1]. Some people use the term *photographic realism* or *photorealism* to refer to a picture that attempts to synthesize the field of light intensities that would be focused on the film plane of a camera aimed at the objects depicted, but the others engaged in non-photorealistic rendering techniques comprehensive drawing techniques rather than accurately simulating optical phenomena [2][3][4].

The creation of realistic pictures is an important goal in fields such as simulation,

design, entertainment and advertising, research and education, and command and control [5].

Designers of 3D objects such as buildings generally want to see how their preliminary designs look so that they can get the best results for their design. Creating realistic computer-generated images is often an easier, less expensive, and more effective way to see preliminary results than building models and prototypes, and also allows the consideration of additional alternative designs. If the design work itself is also computer-based, a digital description of the object may already be available to use in creating the images. Ideally, the designer can also interact with the displayed image to modify the design.

A fundamental difficulty in achieving total visual realism is the complexity of the real world. There are many surface textures, subtle color gradations, shadows, reflections, and slight irregularities in the surrounding objects. Think of patterns on wrinkled cloth, the texture of skin, tousled hair, scuff marks on the floor, and chipped paint on the wall. These all combine to create a real visual experience. The computational costs of simulating these effects can be very high: from many minutes to hours even on powerful computers, But, a more realistic picture is not necessarily a more desirable or useful one, especially if the ultimate goal of a picture is to convey information.

For designing a building it is particularly desirable to display not only one figure of the completed building but also its images of various orientations and aspects. To get best results for every different images of a building, various conditions including lighting, shading, color, material properties, viewing positions and multiple lights positions should be taken into account synthetically. Many people have worked on this topic. For example, in order to get aesthetic effect under various atmospheric conditions, Kaneda *et al.* [6] proposed a method for displaying realistic images of the 3-D objects, i.e. buildings, and particles in the atmosphere, cloud and haze,

under various atmospheric conditions taking account of the spectral distribution of direct sunlight and sky light as an ambient light source. The proposed method can create outdoor images taking account of hue, brightness, and saturation. But, this method has some limitations because it has not taken account of material properties and various aspects and angles of a building.

Generally, for creating totally realistic images, various conditions such as lighting, shading, coloring, viewing and material properties should be taken into account. For example, Nishita [7] presented a method for area light sources. Klassen [8] proposed a method for displaying the color of the sun and the hue of the sky taking into account both scattering and absorption of the sunlight due to air molecules and aerosols in the atmosphere. Inakage [9] improved Klassen's method by approximating geometric optics for large particles such as raindrops, but the method is inadequate for generating realistic images for visual assessment because it does not take into account sky light and specular reflectance.

1.2 OpenGL Interface and Visual Analysis

OpenGL is a software interface that allows graphics programmers to create high-quality 3D graphic images complete with shading, lighting, and other techniques.

OpenGL provides a wide range of graphics features including about 250 routines and approximately 120 distinct commands to draw various primitives such as points, lines, and polygons. It supports shading, texture mapping, antialiasing, lighting, and animation features, and atmospheric effects such as fogging and simulation of depth-of-field. Also, OpenGL provides language binding features so that it can be called from C, C++, Java and other languages. OpenGL is now available on many platforms including Windows 95/NT and UNIX.

Visual Analysis is a powerful software package used by architects for designing and pre-evaluating a building in PC based on Windows 95/NT. It is fast and efficient

even on a small PC. After designers finish the design of a building, the final step is to create realistic images of 3D scenes in various orientations and aspects, such as front, back, left, right, up, down, zoomed in, zoomed out, and random angle and direction rotations, so that the designers can find their shortages and drawbacks of their design and make the best modification. The original rendering subprograms of Visual Analysis didn't take into account lighting, shading, coloring and material properties comprehensively, and can not create realistic images in the final step. Although some aesthetic effect is required, the main purpose is to get correct information from the images rendered. Based on this purpose, the final images can be free of complications of surface textures so that not only more realistic images can be created but also the performance of the whole software package Visual Analysis can be improved.

Visual Analysis was developed by using OpenGL and Borland C++ based on Windows 95/NT platforms.

1.3 Objective and Thesis Outline

Due to the drawbacks of the Visual Analysis software package, the purpose of this thesis is to develop a realistic image rending program by using OpenGL and Borland C++ for Visual Analysis based on Windows 95/NT without reducing the fast rendering performance of the original software package. This thesis presents the efficient empirical methods based on lighting, shading and viewing model. As a result, the architects can get more realistic images of the buildings they have designed and the performance of Visual Analysis will be greatly improved.

The paper is organized as follows. In Chapter 2 we present basic graphics concepts and realistic rendering techniques, In Chapter 3 we describe the OpenGL features and functions that implement these techniques and combine these functions with Visual Analysis. Finally, in Chapter 4, we discuss the experimental testing and

offer concluding remarks.

Chapter 2

Rendering Techniques for Realistic Images

2.1 Introduction

Creating realistic images involves a number of stages, including generating models of the objects, selecting a viewing specification and lighting conditions, and so on. The process of creating images from models is called *rendering*.

There are a series of techniques that make it possible to create successively more realistic pictures. The following sections present some important concepts and techniques in realistic rendering.

2.2 Coloring

Color is an immensely complex subject and it plays an essential role in modern computer graphics. The purpose for using color is not only for aesthetics, but also for creating realistic images of the real world.

A color model is a specification of a 3D color coordinate system and a visible subset in the coordinate system within which all colors in a particular color gamut lie. There are three hardware-oriented color models available. They are RGB (Red, Green, Blue), used with color CRT monitors; YIQ, the broadcast TV color system;

and CMY (Cyan, Magenta, Yellow), used for certain color-printing devices. For each model there is a mean of converting to another specification [11].

Because the goal of this thesis is to create realistic images on the computer color monitor, we only discuss one, the *RGB color model.*

RGB is the primary color type which is used by most of 3D graphics APIs (Application Program Interfaces) , because the RGB color type is used in CRT (Cathode Ray Tube) monitors and color raster graphics employs a *Cartesian Coordinate System.* A black-and-white version of the RGB cube is shown in Figure 2.1. The R, G, and B values can range from 0.0 (none) to 1.0 (full intensity). For example, one vertex is Black (0.0, 0.0, 0.0) and the opposite is brightest White (1.0, 1.0, 1.0). From the black vertex, three edges go to the adjacent vertices: Red, Green and Blue. *Red* corresponds to the X axis, *Green* to the Y axis and *Blue* to the Z axis. At the three vertices adjacent to white are the mixtures of colors. For example, blending Green and Blue creates shades of Cyan (0.0, 1.0, 1.0); Blue and Red combine for Magenta (1.0, 0.0, 1.0); Red and Green create Yellow (1.0, 1.0, 0.0). The Gray scale runs through the main diagonal of the cube from the black vertex to the white vertex.

A computer-graphics monitor emulates visible colors by lighting pixels with a combination of red, green, and blue light in proportions that excite the red-sensitive,

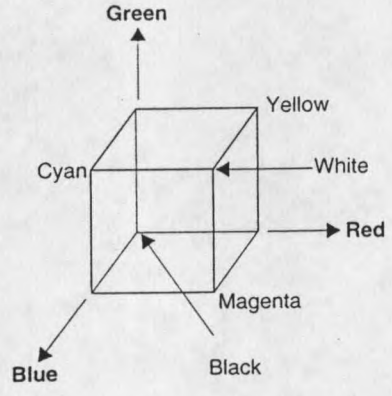


Figure 2.1: The RGB color cube

green-sensitive, and blue-sensitive cones in the retina in such a way that it matches the excitation levels generated by the photon mix it's trying to emulate. To display a particular color, the monitor sends the right amounts of red, green, and blue light to appropriately stimulate the different types of cone cells in human's eye. A color monitor can send different proportions of red, green, and blue to each of the pixels, and the eye sees a million or so pinpoints of light, each with its own color.

2.3 Lighting

In section 2.2, we discussed the color features. In fact, in the real world, each surface's appearance of an object not only depends on color, but also on the types of light sources illuminating it, its other properties (texture, reflectance), and its position and orientation with respect to the light sources, viewer, and other surfaces. Also, part of color computation of each pixel actually depends on what lighting is used in the scene and on how objects in the scene reflect or absorb that light.

Lighting is the process of calculating how much light reaches an object and usually refers to the calculation of how much light is reflected off the object.

2.3.1 Types of Light

There are five types of light commonly used by 3D graphics systems. They are *ambient light*, *directional light* (*infinite light*), *point light*, *spotlight* and *area light*. The *ambient light* simulates the overall amount of light that is presented everywhere in the environment. Consequently, when defining an ambient light, only the intensity and color parameter have to be specified. The *directional light* is considered to be located infinitely far away (simulating, in effect, the great distance of the sun). It has no location, but has direction. For example, from front to back, or from upper left to lower right, and so on. Thus, it specifies sunlight (which comes from a certain direction) more accurately. The *point light* is the most common type of light

in computer graphics systems [12]. It simulates the effect of a bare light located at a specific point in space. It radiates light equally in all directions. The point light is defined by three basic parameters : location, color and intensity. The *spotlight* is similar to the point light, except for restricting the shape of the light it emits to a cone. It can simulate the flash-light effects. The *area light* simulates the effect of an entire area emitting light rather than a single point. It is useful to simulate the bank of fluorescent lights on a ceiling or the screen of a television set.

2.3.2 Reflection Light of Materials

Except for the five light types, another important impact on a real object is the *reflection* of light. There are three basic types of reflection: *ambient reflection*, *diffuse reflection*, and *specular reflection*.

Ambient reflection reflects only ambient light. Since ambient light comes from all directions, ambient reflection is also scattered in all directions. Ambient light is reflected uniformly and with the same intensity from all points on an object's surface, regardless of the position of the object or the orientation or curvature of its surface. Ambient reflection is represented by *the ambient reflection coefficient* which actually is a material property. Along with the other material properties that we will discuss, it may be thought of as characterizing the material from which the surface is made. Like some of the other properties, the ambient-reflection coefficient is an empirical convenience and does not correspond directly to any physical property of real materials but is useful for lighting the dark side of objects. Considering ambient light and reflection, we have the ambient illumination equation [13]:

$$I = I_a K_a \tag{2.1}$$

I_a is the intensity of the ambient light, assumed to be constant for all objects. K_a is the *ambient-reflection coefficient*, the amount of ambient light reflected from an

object's surface.

Diffuse reflection (also called as *Lambertian reflection*) redistributes nonambient light in all directions. The amount of light that bounces off the surface depends on the orientation of the surface with respect to the light, the *diffuse reflection coefficient* and the amount of nonambient light available. The position of the viewer does not affect the amount of diffuse reflection. Dull, matte surfaces, such as chalk and concrete, exhibit diffuse reflection. The intensity of diffuse reflections varies in proportion to the angle at which the light strikes the surface. Considering the point light source's intensity and the material's diffuse-reflection coefficient, we have the diffuse illumination equation [14]:

$$I = I_p K_d cos\theta \tag{2.2}$$

I_p is the point light source's intensity; the material's *diffuse-reflection coefficient* K_d is a constant between 0 and 1 and varies from one material to another. The angle of incidence of the light θ is between the direction $\overline{L}$ to the light source and the surface normal $\overline{N}$. Assuming that the vectors $\overline{N}$ and $\overline{L}$ have been normalized, we can rewrite Eq. (2.2) by using the dot product:

$$I = I_p K_d(\overline{N} \cdot \overline{L}) \tag{2.3}$$

Specular reflection can be observed on any shiny surface. For example, shiny metal or plastic has a high specular component, and chalk or concrete has almost none. Specular reflection produces specular highlights, but these highlights only occur when the angle to the viewer is equal to the angle of reflection. The specular color is computed from the *specular coefficient*, the available light, the object's specular color and the dot product of the reflection vector and the vector to the viewer raised to the specular exponent.

2.3.3 Light-source Attenuation

So far, five types of light and three types of reflection have been discussed. In order to simulate realistic lights, another important factor is the light-source attenuation. We introduce a *light-source attenuation factor*, f_{att} [15]:

$$f_{att} = min\left(\frac{1}{c_1 + c_2 d_L + c_3 d_L^2}, 1\right) \tag{2.4}$$

where, c_1, c_2 and c_3 are user-defined constants associated with the light source and d_L is the distance the light travels from the point source to the surface.

2.3.4 Lighting Model

Taking account into the light, reflection, color and light-source attenuation, we have a popular lighting model [16]:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + f_{att} I_{p\lambda}[k_d O_{d\lambda}(\overline{N} \cdot \overline{L}) + k_s O_{s\lambda}(\overline{R} \cdot \overline{V})^n] \tag{2.5}$$

where, λ represents R, G, or B color; $O_{d\lambda}$ represents an object's diffuse color of color λ; $O_{s\lambda}$ is the object's specular color of color λ; $I_{a\lambda}$ is the intensity of the ambient light of color λ; $I_{p\lambda}$ is the point light source's intensity of color λ; k_a is the ambient-reflection coefficient; k_d is the diffuse-reflection coefficient; k_s is the specular-reflection coefficient; f_{att} is the light-source attenuation factor; $\overline{N}$, $\overline{L}$, $\overline{R}$ and $\overline{V}$ are the surface normal, the direction to the light source, the direction of reflection and the viewpoint direction separately and they are all normalized. n is the material's specular-reflection exponent.

Sometimes we need more than one light source for creating realistic images. If there are m light sources, then the terms for each light source are summed [17]:

$$I_\lambda = I_{a\lambda} k_a O_{d\lambda} + \sum_{1 \le i \le m} f\mathtt{att}_i I\mathrm{p}\lambda_i [k_d O_{d\lambda}(\overline{N} \cdot \overline{L_i}) + k_s O_{s\lambda}(\overline{R_i} \cdot \overline{V})^n] \tag{2.6}$$

So far, the lighting models discussed above are largely the result of a common sense, practical approach to graphics. Cook and Torrance introduced physically based lighting models [18]. Also, several enhancements and generalizations to the Cook-Torrance lighting models have been made [19][20][21][22].

2.4 Shading

Shading is the effect of light on the surface of an object as the surface goes from lighter to darker. In other words, shading is the process of applying different shades of color to different pixels as an object is rendered. If there are several objects in a scene, however, the situation can become more complex, in that each object not only is shaded but might also cast shadows on other objects. It should be clear that we can shade any surface by calculating the surface normal at each visible point and applying the desired lighting model at that point. Unfortunately, this brute-force shading model is expensive.

In order to create realistic images, we must cast shading to simulate real visual effects. In almost all 3D computer graphics systems, the calculation of shading is fairly straightforward. In this section, we describe several more efficient shading models for surfaces defined by polygons and polygon meshes.

2.4.1 Flat Shading

The simplest shading model for a polygon is *flat shading*, also called *faceted shading*, *Lambent shading* or *constant shading*. This approach applies an lighting model once to determine a single intensity value that is then used to shade an entire polygon. This approach is valid if several assumptions are true:

1. The light source is at infinity, so $\overline{N} \cdot \overline{L}$ is constant across the polygon face.

2. The viewer is at infinity, so $\overline{N} \cdot \overline{V}$ is constant across the polygon face.

3. The polygon represents the actual surface being modeled and is not an approximation to a curved surface.

Flat shading results in very fast renderings. However, it has several limitations. It ignores smoothing information. It also can not receive shadows and can not use bump maps or reflection maps [23].

2.4.2 Interpolated Shading

Interpolated shading is a technique in which shading information is computed for each polygon vertex and interpolated across the polygons to determine the shading at each pixel. This method is especially effective when a polygonal object description is intended to approximate a curved surface. In this case, the shading information computed at each vertex can be based on the surface's actual orientation at that point and is used for all of the polygons that share that vertex. Interpolating among these values across a polygon approximates the smooth changes in shading that occur across a curved, rather than planar, surface. Two basic shading models for polygon meshes take advantage of the information provided by adjacent polygons to simulate a smooth surface. They are known as *Gouraud shading* and *Phong shading*. Current 3D graphics workstations typically support one or both of these approaches through a combination of hardware and firmware. OpenGL only supports Gouraud shading.

Gouraud shading [24] is the most common smooth-shading algorithm. Gouraud shading is also called *intensity interpolation shading* or *color interpolation shading* because it eliminates intensity discontinuities by interpolating the intensity for each polygon. First, the Gouraud shading algorithm positions a normal at each vertex of each polygon. This normal is perpendicular to the surface of the polygon. Second, the Gouraud shading algorithm calculates an average of the normals at the given vertex. The vertex normals can be approximated by averaging the surface normals

of all polygonal facets sharing each vertex (see Figure 2.2). If an edge is meant to be visible, then two vertex normals can be found, one of each side of edge, by averaging the normals of polygons on each side of the edge separately. The third step of Gouraud shading is to find vertex intensities, by using the vertex normals with any desired lighting model. Finally, each polygon is shaded by linear interpolation of vertex intensities along each edge and then between edges along each scan line. Gouraud shading is a relatively simple and fast algorithm for smooth and non-smooth objects, but it has some limitations. For example, objects can not receive or cast shadows and can not use bump maps or reflection maps with Gouraud shading.

Phong shading [25] is an expensive algorithm that can produces better results than Gouraud shading. Phong shading, also known as *normal-vector interpolation shading*, interpolates the surface normal vector $\overline{N}$, rather than the intensity. OpenGL doesn't provide the Phong shading model.

Gouraud shading and Phong shading both have their own characteristics. But, they both have some common problems which all interpolated-shading models have, such as polygonal silhouette, perspective distortion, orientation dependence, and unrepresentative vertex normals and problems at shared vertices. See [26] for more details about these problems.

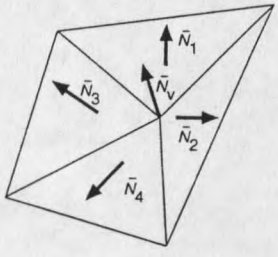


Figure 2.2: Normalized polygon surface normals may be averaged to obtain vertex normals.

2.5 Viewing

The previous sections explained how to draw the geometric objects with colors, lighting and shading effects. In this section, we will discuss how to position and orient objects in three-dimensional space and how to establish the location—also in three-dimensional space—of the viewpoint. All of these factors help determine exactly what image appears on the screen.

There are some basic principles related to the 3D viewing system, such as coordinate system, screen space, projections and the camera analogy, etc. The conceptual model of the 3D viewing process is shown in Figure 2.3.

The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. Thus, a virtual camera is often used as a conceptual aid in computer graphics. The analogy of the film plane in computer graphics is the view plane which is the plane on which the scene is projected. Additionally, the camera is allowed to be rotated about the view direction. The camera, of course, is really just a computer program that produces an image on a display screen, and the object is a 3D dataset comprising a collection of points, lines and surfaces. The virtual camera is a useful concept, but it is not enough to produce an image.

After the virtual camera is specified, the next concept is the projection in the

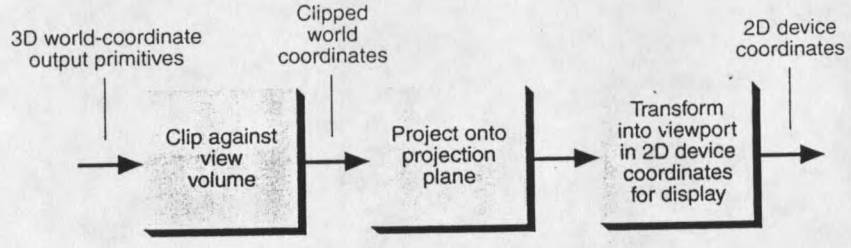


Figure 2.3: Conceptual model of the 3D viewing process

3D viewing system. The projection of a 3D object is defined by straight projection rays, called projectors, emanating from a center of projection, passing through each point of the object, and intersecting a projection plane to form the projection [27]. There are two most important projection classes, *perspective projection* (Figure 2.4) and *parallel orthographic projection* (Figure 2.5) [28].

Perspective projection makes objects that are farther away appear smaller, as you see things in daily life. For example, it makes railroad tracks appear to converge in the distance.

Parallel orthographic projection maps objects directly onto the screen without affecting their relative size. Parallel orthographic projection is used in architectural and computer-aided design applications where the final image needs to reflect the measurements of objects rather than how they might look. The center of projection and *direction of projection* (DOP) are defined by a *projection reference point* (PRP) and an indicator of the projection type. If the projection type is perspective, then PRP is the center of projection. If the projection type is parallel, then the DOP is from the PRP to CW (*Center of Window*).

The *view plane*, also called the *projection plane*, is defined by a point on the plane called the *view reference point* (VRP) and a normal to the plane called the *view-plane normal* (VPN). The view plane may be anywhere with respect to the world objects to be projected. It may be in front of, cut through, or be behind the

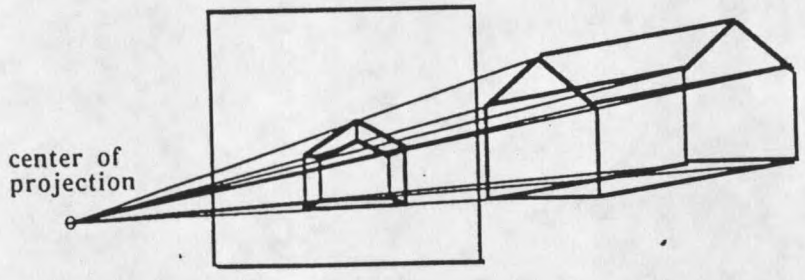


Figure 2.4: Principle of Perspective Projection

objects.

There are several coordinate systems available on current computer graphics systems, but the most commonly used are the *world coordinate system*, *view-reference coordinate system* and *normalized-projection coordinate system*. Figure 2.6 [29] shows how they are related to each other in the 3D viewing system.

The world coordinate (WC) system is the most commonly used system, and is easy to understand. The *view-reference coordinate* (VRC) system, used to defined a view volume, is not as easy as the world coordinate system. The *normalized-projection coordinate* (NPC) system is a coordinate system which is used to normalize the view volume in some pre-specified range.

Although the viewing system is more complex than lighting and shading in 3D graphics system, it is a necessary part for a 3D graphics API which is used to create realistic images on a screen.

2.6 Other Techniques

In addition to all the important concepts we have discussed, we still need some other techniques for creating realistic images. For example, realism is further enhanced if the material properties of each object are taken into account when its shading is determined; *Texture mapping* not only provides additional depth cues, but also

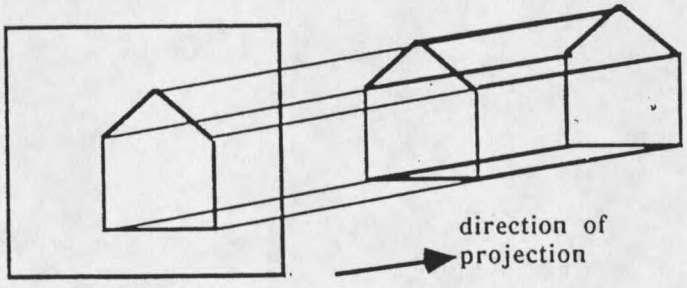


Figure 2.5: Principle of Parallel Projection

can mimic the surface detail of real objects; Reproducing *shadows* cast by objects on one another can introduce further realism; *Transparency* is also very useful in picture making for transparent surface. For more details about these techniques, see Foley *et al*'s *Introduction To Computer Graphics* [30].

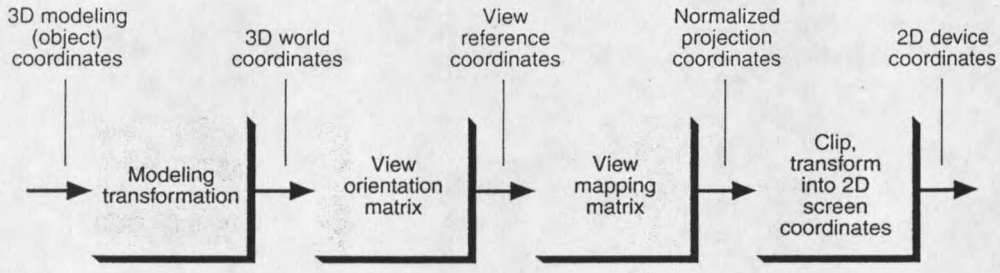


Figure 2.6: Coordinate systems and how they relate to one another

Chapter 3

Visual Analysis and OpenGL Implementation

3.1 Introduction

Visual Analysis is a complicated but efficient software package which can be used by architects to design a building. This software package was designed with OpenGL and Borland C++ based on Windows 95/NT platforms. Architects can enter the data for a building, select various materials and material properties, and create realistic images for the designed building from various vantage points.

Although Visual Analysis is a useful tool for designing building, it lacks the ability for creating realistic pictures in the final step. It can only create rough pictures with lines and polygons represented by several unrealistic colors.

Creating realistic pictures involves a number of stages and many techniques that are discussed in Chapter 2. Because the designed buildings mostly are built from flat polygons, creating totally realistic images from various viewing points is very difficult. In this chapter, we will presents realistic rendering techniques in Visual Analysis from a variety of perspectives, including coloring, lighting, shading and viewing techniques that discussed in Chapter 2, by using OpenGL and Borland C++ based on Windows 95/NT platforms.

3.2 Coloring in Visual Analysis

In section 2.1, we have discussed the general coloring concepts. In this section, we will discuss coloring in OpenGL and implementation in Visual Analysis.

One goal of almost all OpenGL applications is to draw color pictures in a window on the screen. The window is a rectangular array of pixels, each of which contains and displays its own color. Thus, in a sense, the point of all the calculations performed by an OpenGL implementation–calculations that take into account OpenGL commands, state information, and values of parameters–is to determine the final color of every pixel that's to be drawn in the window. OpenGL provides two display color modes: *RGBA mode* and *color-index mode.*

In either color-index or RGBA mode, a certain amount of color data is stored at each pixel. This amount is determined by the number of bitplanes in the frame buffer. A bitplane contains one bit of data for each pixel. To find out the number of bitplanes available for red, green, blue, alpha, or color-index values, we can use glGetIntegrev() with GL_RED_BITS, GL_GREEN_BITS, GL_BLUE_BITS, GL_ALPHA_BITS, and GL_INDEX_BITS.

In RGBA mode, the hardware sets aside a certain number of bitplanes for each of the R, G, B and A components. The A in RGBA means the *alpha* value. It has no direct effect on the color displayed on the screen. It can be used for many things, including blending and transparency, and it can have an indiect effect on the value of R, G and B. The alpha value is between 0.0 which is the minimum intensity and 1.0 which specifies the maximum intensity, and it isn't specified in color index mode. The alpha component is never displayed directly. It's typically used to control color blending. By convention, OpenGL alpha corresponds to the notion of opacity rather than transparency, meaning that an alpha value of 1.0 implies complete opacity, and an alpha value of 0.0 implies complete transparency.

In color-index mode, OpenGL uses a color map, which provides indices where the primary red, green, and blue values can be mixed. The system stores the color index in the bitplanes for each pixel. Then those bitplane values reference the color map, and the screen is painted with corresponding red, green, and blue values from the color map.

In general, one should use RGBA mode: it works with texture mapping and works better with lighting, shading, fog, antialiasing and blending. RGBA mode provides more flexibility than color-index mode. Thus, in Visual Analysis, we choose RGBA mode, and can use the glColor*() command to select a current color in OpenGL.

3.3 Lighting and shading in Visual Analysis

In sections 2.3 and 2.4, we have discussed the basic principles of lighting and shading. In this section, we will discuss how to control the lighting and shading in a scene and how to choose appropriate values to create realistic images in Visual Analysis.

OpenGL provides four types of light source: *ambient*, *diffuse*, *specular* and *position*. 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 OpenGL lighting model makes the approximation that a material's color depends on the percentages of the incoming red, green, and blue light it reflects. OpenGL provides three types of reflectances for materials: ambient, diffuse and specular reflectances.

The color components specified for lights mean something different than for materials. For a light, the numbers correspond to a percentage of full intensity for each color. For materials, the numbers correspond to the reflected proportions of those colors. In other words, if an OpenGL light has components (LR, LG, LB), and

a material has corresponding components (MR, MG, MB), then, ignoring all other reflectivity effects, the light that arrives at the eye is given by (LR*MR, LG*MG, LB*MB). If we have two lights, which send (R1, G1, B1) and (R2, G2, B2) to the eye, OpenGL adds the components, giving (R1+R2, G1+G2, B1+B2).

3.3.1 Creating Light Sources

Light sources have a number of properties, such as color, position, and direction. The OpenGL command used to specify all properties of lights is glLight*(); it takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

Command format:

void **glLight**if[v](Glenum *light*, Glenum *pname*, *TYPE param*);

It creates the light specified by *light*, which can be GL_LIGHT0, GL_LIGHT1, ..., or GL_LIGHT7. The characteristic of the light being set is defined by *pname*, which specifies a named parameter (see Table 3.1). The *param* argument indicates the values to which the *pname* characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

The default values listed for GL_DIFFUSE and GL_SPECULAR in table apply only to GL_LIGHT0. For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR.

OpenGL provides three different color-related parameters, GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR with any particular light. The GL_AMBIENT parameter refers to the RGBA intensity of the ambient light that a particular light source adds to the scene. The default value means that there is no ambient light since GL_AMBIENT is (0.0, 0.0, 0.0, 1.0). In Visual Analysis, we experimented the many ambient values for GL_LIGHT0, and finally we chose (0.5, 0.5, 0.5, 1.0) as

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation

Table 3.1: Default values for *pname* parameter of glLight*()

GL_AMBIENT value. The GL_DIFFUSE parameter defines the RGBA color of diffuse light that a particular light source adds to a scene. By default, GL_DIFFUSE is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0, which produces a bright, white light. In Visual Analysis, we select (0.8, 0.8, 0.8, 1.0) as GL_DIFFUSE parameters for GL_LIGHT0. It produces a not very bright and white light. The GL_SPECULAR parameter affects the color of the specular highlight on an object. By default, GL_SPECULAR is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0. In Visual Analysis, we tested the many values and finally choose (0.3, 0.3, 0.3, 1.0) as the GL_SPECULAR parameter as it can create the most effective on realistic pictures. We use the following commands for GL_LIGHT0 in Visual Analysis:

```
GLfloat glfLightAmbient[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat glfLightDiffuse[] = { 0.8f, 0.8f, 0.8f, 1.0f };
GLfloat glfLightSpecular[] = { 0.3f, 0.3f, 0.3f, 1.0f };
glLightfv( GL_LIGHT0, GL_AMBIENT, glfLightAmbient );
glLightfv( GL_LIGHT0, GL_DIFFUSE, glfLightDiffuse );
glLightfv( GL_LIGHT0, GL_SPECULAR, glfLightSpecular );
```

After the light color has been set, the light position and attenuation need to be fixed. In OpenGL, there are two light sources: *directional light source* and *positional light source*. As we discussed in section 2.3, the directional light source is located infinitely far away (simulating, in effect, the great distance of the sun). Thus, it has no position. The positional light has an exact position within the scene. A desk lamp is an example of a positional light source. In OpenGL, the light position can be defined as follows:

```
GLfloat light_position[] = { x, y, z, w };
glLightfv ( GL_LIGHT0, GL_POSITION, light_position );
```

As shown, we supply a vector of four values (x, y, z, w) for the GL_POSITION parameter. If the last value, w, is zero, the corresponding light source is a directional one, and the (x, y, z) value describes its direction. If the w value is nonzero, the light is positional, and the (x, y, z) values specify the location of the light in homogeneous object coordinates. Generally, a positional light radiates in all directions, but we can restrict it to producing a cone of illumination by defining the light as a spotlight.

In Visual Analysis, first, we use directional light for GL_LIGHT0, and use view direction as its light direction. The commands are as follows:

```
double dir[ 3 ];
mView.GetViewDirection( dir );
GLfloat light_pos[ 4 ];
light_pos[ 0 ] = dir[ 0 ] * mViewDistance / mView.ViewScale * 1.05;
light_pos[ 1 ] = dir[ 1 ] * mViewDistance / mView.ViewScale * 1.05;
light_pos[ 2 ] = dir[ 2 ] * mViewDistance / mView.ViewScale * 0.85;
light_pos[ 3 ] = 0.0;
glLightfv (GL_LIGHT0, GL_POSITION, light_pos );
```

mViewDistance and mView.ViewScale are view distance and view scale values which are calculated in other subprograms. From the commands above, we know that the viewing direction always points into the screen and the light is far away behind the eye. Then, we change the value of light_pos [3] into 1.0, meaning that we use the positional light source and the light position is just behind the eye.

For real world lights, the intensity decreases as distance from the light increases. Since a directional light is infinitely far away, it doesn't make sense to attenuate its intensity over distance, so attenuation is disabled for a directional light. However, the positional light has to be attenuated for creating realistic effects. OpenGL attenuates a light source by multiplying the contribution of that source by an attenuation factor:

$$Attenuation factor = \frac{1}{k_c + k_l d + k_q d^2} \tag{3.1}$$

where,

d = distance between the light's position and the vertex

k_c = GL_CONSTANT_ATTENUATION

k_e = GL_LINEAR_ATTENUATION

k_q = GL_QUADRATIC_ATTENUATION

By default, k_c is 1.0 and both k_e and k_q are zero. In Visual Analysis, we only changed the GL_CONSTANT_ATTENUATION parameter value. k_e and k_q are still zero. We use the parameter mView.ViewScale to calculate k_c. The program segment is:

```
GLfloat atten;
if ( mView.ViewScale > 1.0 )
   atten = pow( 1.0 / mView.ViewScale, 1.0 / 3.0 );
else
```

```
    atten = 1.0 /mView.ViewScale;
  glLightf( GL_LIGHT0, GL_CONSTANT_ATTENUATION, atten );
```

where atten $= k_c$. It is calculated dynamically each time the Visual Analysis draws the picture and it is always greater or equal than 1.0. Because the effect of using directional light is very unreal, we don't use it in Visual Analysis and always use positional light and intensity attenuation. Figures 3.1 to 3.7 show various orientations of the same building.

In OpenGL, we can have many lights (at least eight lights) to improve the realistic effects of the images. Since OpenGL needs to perform calculations to determine how much light each vertex receives from each light source, increasing the number of lights adversely affects performance [31]. In Visual Analysis, five lights (GL_LIGHT1, GL_LIGHT2, GL_LIGHT3, GL_LIGHT4 and GL_LIGHT0, which was defined previously) were used according to the results of many experiments. For the purpose of simplifying the calculations, the parameters used for GL_LIGHT1 to GL_LIGHT4 were similar to the parameters used for GL_LIGHT0. The propram segments are as follows:

```
GLfloat glfLightAmbient[] = { 0.5f, 0.5f, 0.5f, 1.0f };
GLfloat glfLightDiffuse[] = { 0.8f, 0.8f, 0.8f, 1.0f };
GLfloat glfLightSpecular[] = { 0.3f, 0.3f, 0.3f, 1.0f };
glLightfv (GL_LIGHT1, GL_AMBIENT, glfLightAmbient);
glLightfv (GL_LIGHT1, GL_DIFFUSE, glfLightDiffuse);
glLightfv (GL_LIGHT1, GL_SPECULAR, glfLightSpecular);
glLightfv (GL_LIGHT2, GL_AMBIENT, glfLightAmbient);
glLightfv (GL_LIGHT2, GL_DIFFUSE, glfLightDiffuse);
glLightfv (GL_LIGHT2, GL_SPECULAR, glfLightSpecular);
glLightfv (GL_LIGHT3, GL_AMBIENT, glfLightAmbient);
```

```
glLightfv (GL_LIGHT3, GL_DIFFUSE, glfLightDiffuse);
glLightfv (GL_LIGHT3, GL_SPECULAR, glfLightSpecular);
glLightfv (GL_LIGHT4, GL_AMBIENT, glfLightAmbient);
glLightfv (GL_LIGHT4, GL_DIFFUSE, glfLightDiffuse);
glLightfv (GL_LIGHT4, GL_SPECULAR, glfLightSpecular);
double dir[3];
mView.GetViewDirection( dir );
GLfloat light1_pos[4];
light1_pos[0] = -dir[0]*mViewDistance/mView.ViewScale*1.05;
light1_pos[1] = -dir[1]*mViewDistance/mView.ViewScale*1.05;
light1_pos[2] = -dir[2]*mViewDistance/mView.ViewScale*0.85;
light1_pos[3] = 1.0;
glLightfv( GL_LIGHT1, GL_POSITION, light1_pos );
GLfloat light2_pos[4];
light2_pos[0] = -dir[0]*mViewDistance/mView.ViewScale*1.05;
light2_pos[1] = dir[1]*mViewDistance/mView.ViewScale*1.05;
light2_pos[2] = dir[2]*mViewDistance/mView.ViewScale*0.85;
light2_pos[3] = 1.0;
glLightfv( GL_LIGHT2, GL_POSITION, light2_pos );
GLfloat light3_pos[4];
light3_pos[0] = dir[0]*mViewDistance/mView.ViewScale*1.05;
light3_pos[1] = -dir[1]*mViewDistance/mView.ViewScale*1.05;
light3_pos[2] = dir[2]*mViewDistance/mView.ViewScale*0.85;
light3_pos[3] = 1.0;
glLightfv( GL_LIGHT3, GL_POSITION, light3_pos );
GLfloat light4_pos[4];
light4_pos[0] = dir[0]*mViewDistance/mView.ViewScale*1.05;
```

```
light4_pos[1] = dir[1]*mViewDistance/mView.ViewScale*1.05;
light4_pos[2] = -dir[2]*mViewDistance/mView.ViewScale*0.85;
light4_pos[3] = 1.0;
glLightfv( GL_LIGHT4, GL_POSITION, light4_pos );
GLfloat atten;
if( mView.ViewScale > 1.0 )
    atten = pow( 1./mView.ViewScale, 1./3. );
else
    atten = 1./mView.ViewScale;
glLightf( GL_LIGHT1, GL_CONSTANT_ATTENUATION, atten );
glLightf( GL_LIGHT2, GL_CONSTANT_ATTENUATION, atten );
glLightf( GL_LIGHT3, GL_CONSTANT_ATTENUATION, atten );
glLightf( GL_LIGHT4, GL_CONSTANT_ATTENUATION, atten );
glLightf( GL_LIGHT1, GL_LINEAR_ATTENUATION, atten );
glLightf( GL_LIGHT2, GL_LINEAR_ATTENUATION, atten );
glLightf( GL_LIGHT3, GL_LINEAR_ATTENUATION, atten );
glLightf( GL_LIGHT4, GL_LINEAR_ATTENUATION, atten );
```

Note that all five lights have the same GL_AMBIENT, GL_DIFFUSE, GL_SPECULAR values, but have different positions and directions. All of them are positional light sources and have attenuation, but GL_LIGHT1 to GL_LIGHT4 also have the GL_LINEAR_ATTENUATION parameters in addition to the GL_CONSTANT_ATTENUATION parameters.

3.3.2 Selecting a Global Lighting Model

OpenGL's notion of a lighting model has three components:

1. The global ambient light intensity;

2. Whether the viewpoint position is local to the scene or whether it should be considered to be an infinite distance away;

3. Whether lighting calculations should be performed differently for both the front and back faces of objects.

In addition to the ambient light contributed by each light source, there can be other ambient light that's not from any particular source. This is called the *global ambient light.* To specify the RGBA intensity of such light, use the GL_LIGHT_MODEL_AMBIENT parameter in Visual Analysis:

```
GLfloat lmodel_ambient[] = { 0.3, 0.3, 0.3, 1.0 };
glLightModelfv( GL_LIGHT_MODEL_AMBIENT, lmodel_ambient );
```

since these values yield a small amount of white ambient light. Even if we don't add a specific light source to the scene, we can still see the objects in the scene.

The location of the viewpoint affects the calculations for highlights produced by specular reflectance [32]. More specifically, the intensity of the highlight at a particular vertex depends on the normal at that vertex, the direction from the vertex to the light source, and the direction from the vertex to the viewpoint. With an infinite viewpoint, the direction between it and any vertex in the scene remains constant. A local viewpoint tends to yield more realistic results, but since the direction has to be calculated for each vertex, overall performance is decreased with a local viewpoint. By default, an infinite viewpoint is assumed. In Visual Analysis, we change it to a local viewpoint:

```
glLightModeli( GL_LIGHT_MODEL_LOCAL_VIEWER, GL_TRUE );
```

In Visual Analysis, we also use the two-sided lighting model for all polygons because it can improve the illumination effects of the back-facing polygons. The command is:

```
glLightModeli( GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE );
```

After all these lighting models have been set, we need to explicitly enable the lighting. If lighting isn't enabled, the current color is simply mapped onto the current vertex, and no calculations concerning normals, light source, the lighting model, and material properties are performed. To enable the lighting, we use following command:

```
glEnable( GL_LIGHTING );
```

We also need to enable the five light sources we defined by using following commands:

```
glEnable( GL_LIGHT0 );
glEnable( GL_LIGHT1 );
glEnable( GL_LIGHT2 );
glEnable( GL_LIGHT3 );
glEnable( GL_LIGHT4 );
```

3.3.3 Material Properties in Visual Analysis

In OpenGL, there are five material properties that need to be defined for creating realistic images. They are the ambient color, the diffuse color, the specular color, the shininess, and the color of any emitted light.

The command for material properties is glMaterial*():

void **glMaterialif**[v](GLenum *face*, GLenum *pname*, *TYPE param*);

which specifies a current material property for use in lighting calculations. The *face* parameter can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by *pname* and the desired values for that property are

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0, 1, 1)	ambient, diffuse, and specular color indices

Table 3.2: Default values for *pname* parameter of glMaterial*()

given by *param*, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector versions is used). The nonvector version works only for setting GL_SHININESS. The possible values for *pname* are shown in Table 3.2.

Like the lighting model, we also choose GL_FRONT_AND_BACK as the *face* parameter because the back faces of the polygons of objects might be seen. Then we need to select diffuse and ambient reflection parameters. The GL_DIFFUSE and GL_AMBIENT parameters set with glMaterial*() affect the color of the diffuse and ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what color of an object can be perceived. It is affected by the color of the incident diffuse light and the angle of the incident light relative to the normal direction. The position of the viewpoint doesn't affect diffuse reflectance at all. Ambient reflectance affects the overall color of the object. Because diffuse reflectance is brightest where an object is directly illuminated, ambient reflectance is most noticeable where an object receives no direct illumination. An object's total ambient reflectance is affected by the global ambient light and ambient light from individual light sources. Like diffuse reflectance, ambient reflectance is not affected by the position of the viewpoint. Specular reflection from an object produces highlights. Unlike ambient and diffuse reflection, the amount of specular reflection

seen by a viewer does depend on the location of the viewpoint–it is brightest along the direct angle of reflection. The RGBA color of a specular highlight can be set with GL_SPECULAR and the size and brightness of the highlight can be controlled with GL_SHININESS. Users can assign a number in the range of [0.0, 128.0] to GL_SHININESS– the higher the value, the smaller and brighter the highlight. By specifying an RGBA color for GL_EMISSION, we can make an object appear to be giving off light of that color. Generally, we can use this feature to simulate lamps and other light sources in a scene.

Visual Analysis is generally used for designing a building which is built with materials including steel, aluminum, concrete, masonry and wood. Also, the surfaces of the building structure are mostly flat. Because there are no lamps in the building, we don't need to set the GL_EMISSION parameter. Selecting other material parameters for steel, aluminum, concrete, masonry and wood is the most difficult part in Visual Analysis because most are empirical. After testing many parameter data, we finally got the best data for all those materials. The program segments are as follows:

```
case MASONRY:
{
   GLfloat glfMaterialAmbient[] = { 0.2125, 0.1275, 0.054, 1.0 };
   GLfloat glfMaterialDiffuse[] = { 0.614, 0.4284, 0.38144, 1.0 };
   GLfloat glfMaterialSpecular[] = { 0.493548, 0.41906, 0.466721, 1.0 };
   GLfloat glfMaterialShininess = 10.0;
   glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, glfMaterialAmbient );
   glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, glfMaterialDiffuse );
   glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, glfMaterialSpecular );
   glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, glfMaterialShininess );
   break;
```

```
}
case CONCRETE:
{
   GLfloat glfMaterialAmbient[] = { 0.4, 0.4, 0.4, 1.0 };
   GLfloat glfMaterialDiffuse[] = { 0.6, 0.6, 0.6, 1.0 };
   GLfloat glfMaterialSpecular[] = { 0.0, 0.0, 0.0, 1.0 };
   GLfloat glfMaterialShininess = 10.0;
   glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, glfMaterialAmbient );
   glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, glfMaterialDiffuse );
   glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, glfMaterialSpecular );
   glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, glfMaterialShininess );
   break;
}
case ALUMINUM:
{
   GLfloat glfMaterialAmbient[] = { 0.5, 0.5, 0.5, 1.0 };
   GLfloat glfMaterialDiffuse[] = { 0.8, 0.8, 0.8, 1.0 };
   GLfloat glfMaterialSpecular[] = { 0.7, 0.7, 0.8, 1.0 };
   GLfloat glfMaterialShininess = 128.0;
   glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, glfMaterialAmbient );
   glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, glfMaterialDiffuse );
   glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, glfMaterialSpecular );
   glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, glfMaterialShininess );
   break;
}
case WOOD:
{
```

```
    GLfloat glfMaterialAmbient[] = { 0.6, 0.5, 0.1, 1.0 };
    GLfloat glfMaterialDiffuse[] = { 0.6, 0.5, 0.1, 1.0 };
    GLfloat glfMaterialSpecular[] = { 0.0, 0.0, 0.0, 1.0 };
    GLfloat glfMaterialShininess = 10.0;
    glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, glfMaterialAmbient );
    glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, glfMaterialDiffuse );
    glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, glfMaterialSpecular );
    glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, glfMaterialShininess );
    break;
}
case STEEL:
{
    GLfloat glfMaterialAmbient[] = { 0.5, 0.1, 0.1, 1.0 };
    GLfloat glfMaterialDiffuse[] = { 0.5, 0.1, 0.1, 1.0 };
    GLfloat glfMaterialSpecular[] = { 0.8, 0.3, 0.3, 1.0 };
    GLfloat glfMaterialShininess = 80.0;
    glMaterialfv( GL_FRONT_AND_BACK, GL_AMBIENT, glfMaterialAmbient );
    glMaterialfv( GL_FRONT_AND_BACK, GL_DIFFUSE, glfMaterialDiffuse );
    glMaterialfv( GL_FRONT_AND_BACK, GL_SPECULAR, glfMaterialSpecular );
    glMaterialf ( GL_FRONT_AND_BACK, GL_SHININESS, glfMaterialShininess );
    break;
}
```

By using these parameters combined with the lighting model and positional lights discussed earlier, we can get enhanced realistic images in Visual Analysis. Figures 3.8 to 3.13 show the result of all these materials. Note that, from these figures, we can conclude that the masonry, concrete and aluminum are almost real, but the wood looks a little unreal because generally the wood has textured surfaces.

Because in most cases, the steel is often painted with dark red paint, we choose dark red color and some shininess for steel and it looks better.

3.3.4 The Lighting Equation in OpenGL

In Chapter 2, we have discussed the general equations of lighting model. In this section, we present the equation used by OpenGL to perform lighting calculations to determine colors when in RGBA mode. The entire lighting equation in RGBA mode is as follows:

$$\begin{aligned} vertex\ color \;=\; & emission_{material} + \\ & ambient_{light\ model} * ambient_{material} + \\ & \sum_{i=0}^{n-1} \left(\frac{1}{k_c + k_l d + k_q d^2} \right)_i (spotlighteffect)_i \\ & [ambient_{light} * ambient_{material} + \\ & (max\mathbf{l} \cdot \mathbf{n}, 0) * diffuse_{light} * diffuse_{material} + \\ & (max\mathbf{s} \cdot \mathbf{n}, 0)^{shininess} * specular_{light} * specular_{material}]_i \end{aligned} \tag{3.2}$$

where, the material emission term $emission_{material}$ is the RGB value assigned to the GL_EMISSION parameter. The second term is computed by multiplying the global ambient light by the material's ambient property:

$ambient_{light\ model} * ambient_{material}$

The third term is the most complicated one, which is the contributions from light sources. For more details about its parameters, see [33].

According to this lighting equation, we can select more accurate values for all parameters we discussed in the previous sections because we have a better idea of how the values of parameters affect a vertex's color. Even so, we still need to experiment a lot to create the best effects for the designed buildings.

3.3.5 Shading in OpenGL

We have discussed several shading in section 2.4. Since the design strategy is simple and fast to render a scene in OpenGL, OpenGL only supports two types of shading: *flat shading* and *Gouraud shading*(*smooth shading*). It doesn't support *Phong shading* because Phong shading is considerably more expensive to implement.

With flat shading, the intensity of one vertex of a primitive is duplicated across all of the primitive's vertices. With Gouraud shading (smooth shading), the color at each vertex is treated individually. For a line primitive, the colors along the line segment are interpolated between the vertex colors. For a polygon primitive, the colors for interior of the polygon are interpolated between the vertex colors.

To get realistic effects in Visual Analysis, we specify Gouraud shading (smooth shading) by using command glShadeModel():

```
glShadeModel( GL_SMOOTH );
```

3.4 Viewing in OpenGL

In Chapter 2, we discussed the basic viewing principles. In this section, we discuss how to position and orient models in three-dimensional space and how to establish the location of the viewpoint in OpenGL.

The point of computer graphics is to create a two-dimensional image of three-dimensional objects. There are three computer operations that convert an object's three dimensional coordinates to a pixel position on the screen:

1. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, we use a combination of several transformations to draw a scene.

2. Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.

3. Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a viewport transformation.

In OpenGL, the viewing transformation is analogous to positioning and aiming a camera. Specifying the projection transformation is like choosing a lens for a camera. Two basic types of projections are provided by OpenGL, along with several corresponding commands for describing the relevant parameters in different ways. One type is *perspective projection*, which makes objects that are farther away appear smaller. Because in Visual Analysis, we need to make realistic pictures, we should choose perspective projection. The other type of projection is *orthographic*, which maps objects directly onto the screen without affecting their relative size. We will not discuss orthographic projection because we don't use it in Visual Analysis.

With perspective projection, objects that fall within the viewing volume are projected toward the apex of the pyramid, where the camera or viewpoint is. Objects that are closer to the viewpoint appear larger because they occupy a proportionally larger amount of the viewing volume than those that are farther away, in the larger part of the frustum. This method of projection is commonly used for animation, visual simulation, and any other applications that strive for some degree of realism because it's similar to how our eye (or a camera) works.

There are two commands used for projection transformations, glFrustum() and gluPerspective() [34]. We only discuss the latter one because it's very intuitive to use and we use it in Visual Analysis. The command is as follows:

void **gluPerspective**(GLdouble *fovy*, GLdouble *aspect*, GLdouble *zNear*, GLdou-

ble *zFar*);

It creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. The *fovy* argument is the angle of the field of view in the x-z plane; its value must be in the range [0.0, 180.0]. The *aspect* ratio is the width of the frustum divided by its height. The *zNear* and *zFar* values are the distances between the viewpoint and the clipping plans, along the negative z-axis. They should always be positive (Figure 3.14).

In Visual Analysis, the parameters for gluPerspective() are as follows:

```
gluPerspective( 30.0,        // Field-of-view angle
                mGldAspect, // Aspect ration of viewing volume
                1.0,         // Distance to near clipping plane
                9999999.0); // Distance to far clipping plane
```

Where, mGldAspect is calculated in other subprograms. Figures 3.15 to 3.16 show the perspective viewing effects when the image is zoomed in or zoomed out.

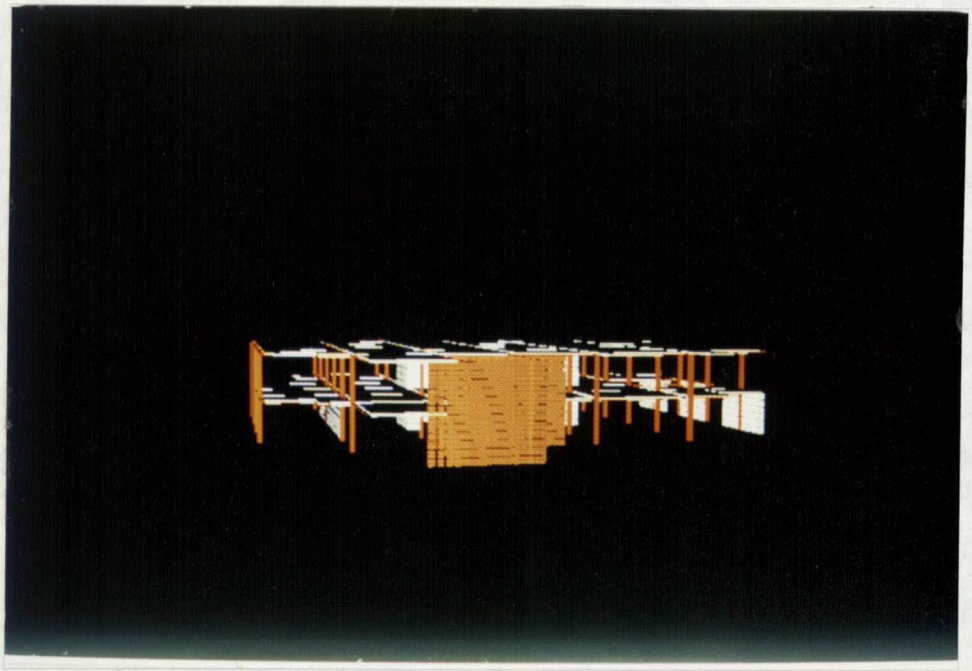


Figure 3.1: Cobleigh Hall (1)

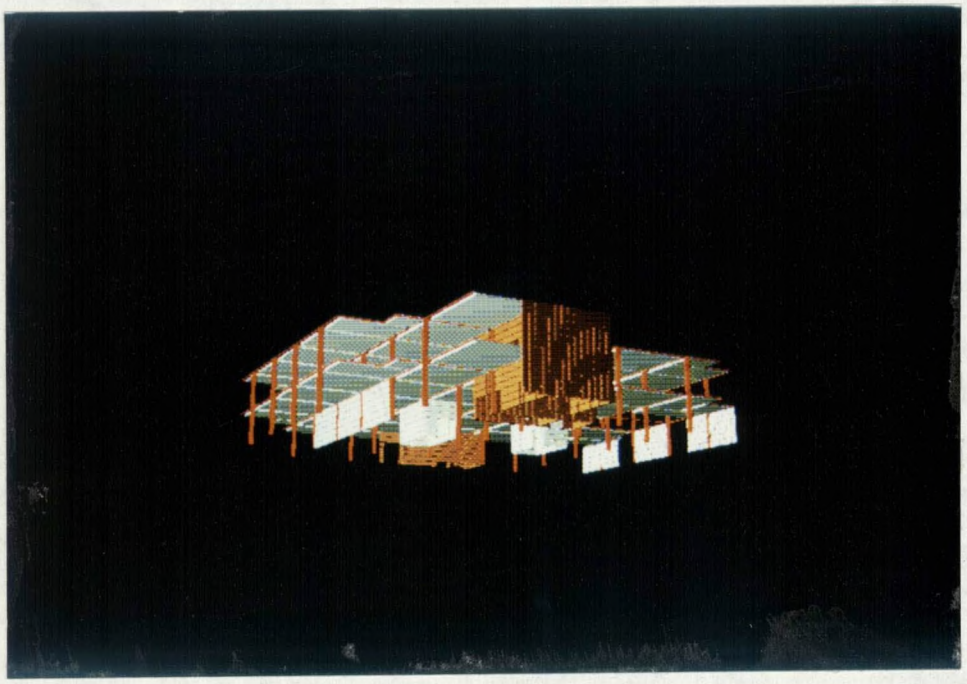


Figure 3.2: Cobleigh Hall (2)

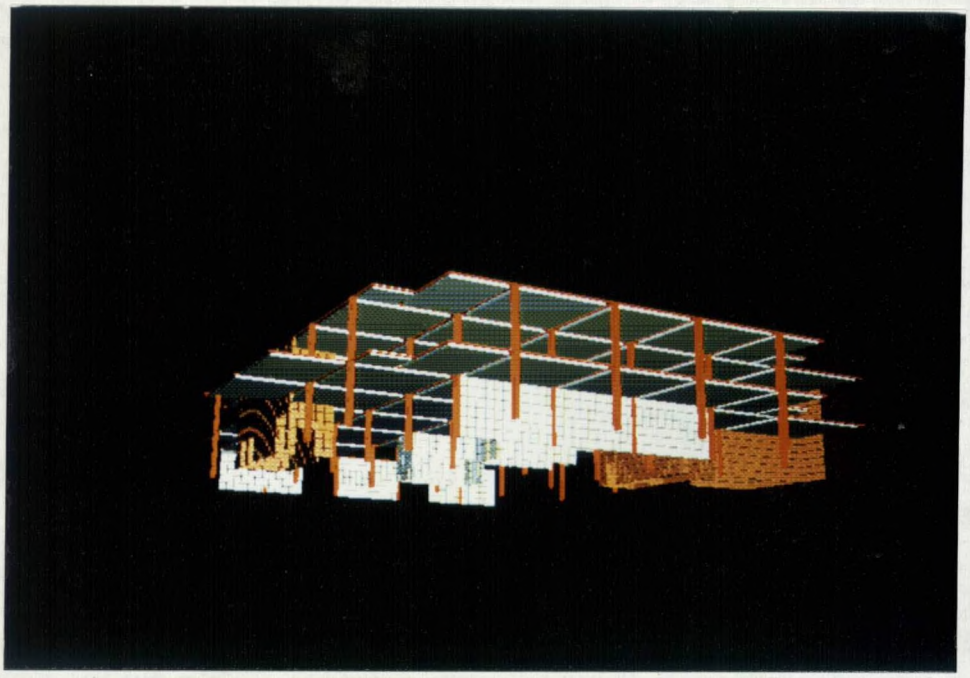


Figure 3.3: Cobleigh Hall (3)

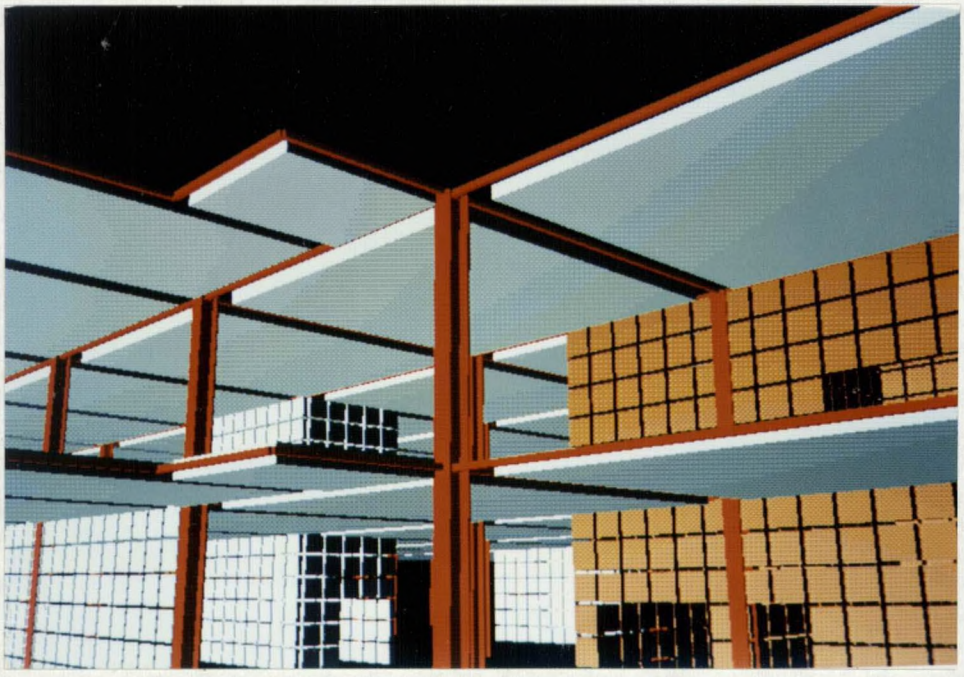


Figure 3.4: Cobleigh Hall (4)

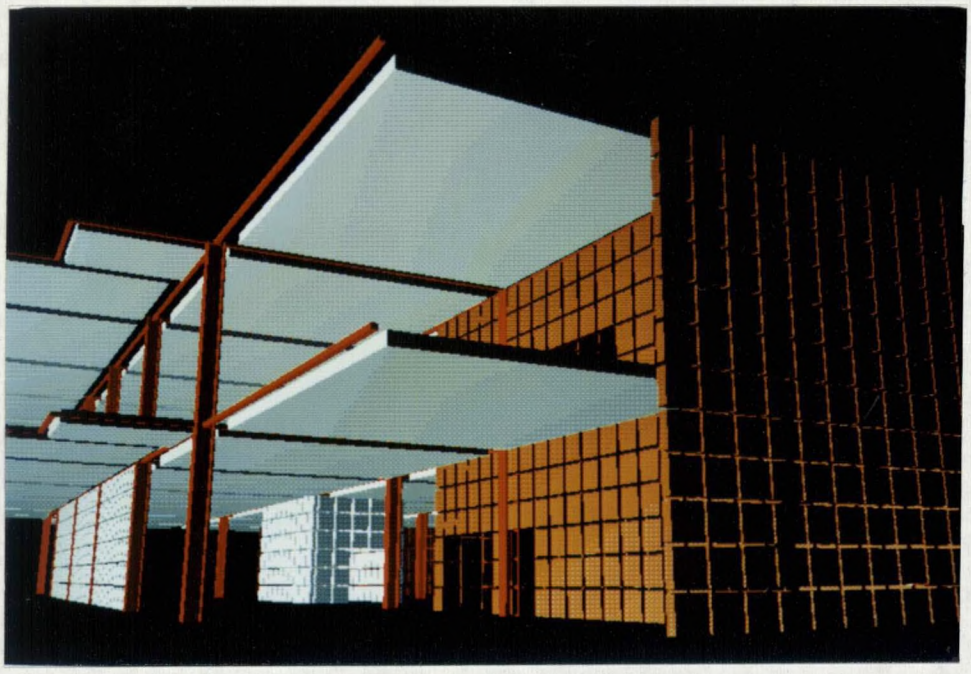


Figure 3.5: Cobleigh Hall (5)

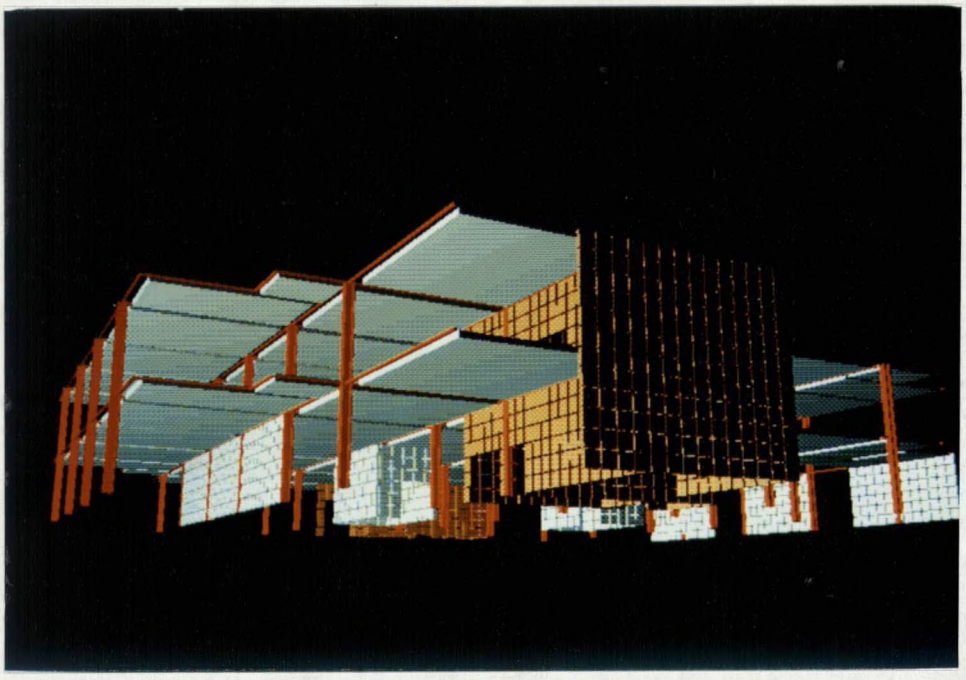


Figure 3.6: Cobleigh Hall (6)

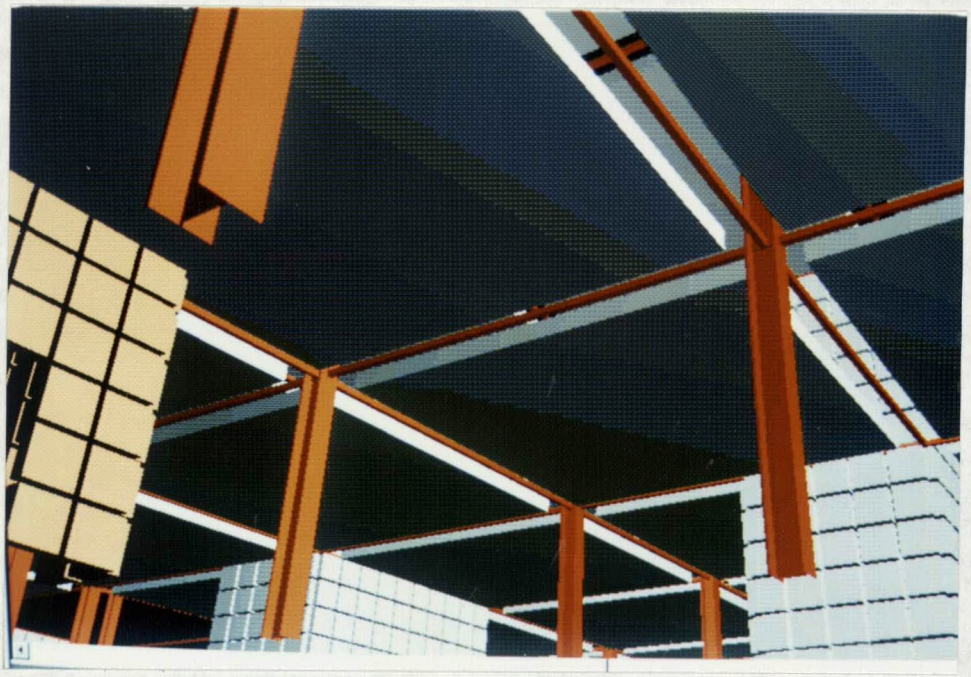


Figure 3.7: Cobleigh Hall (7)

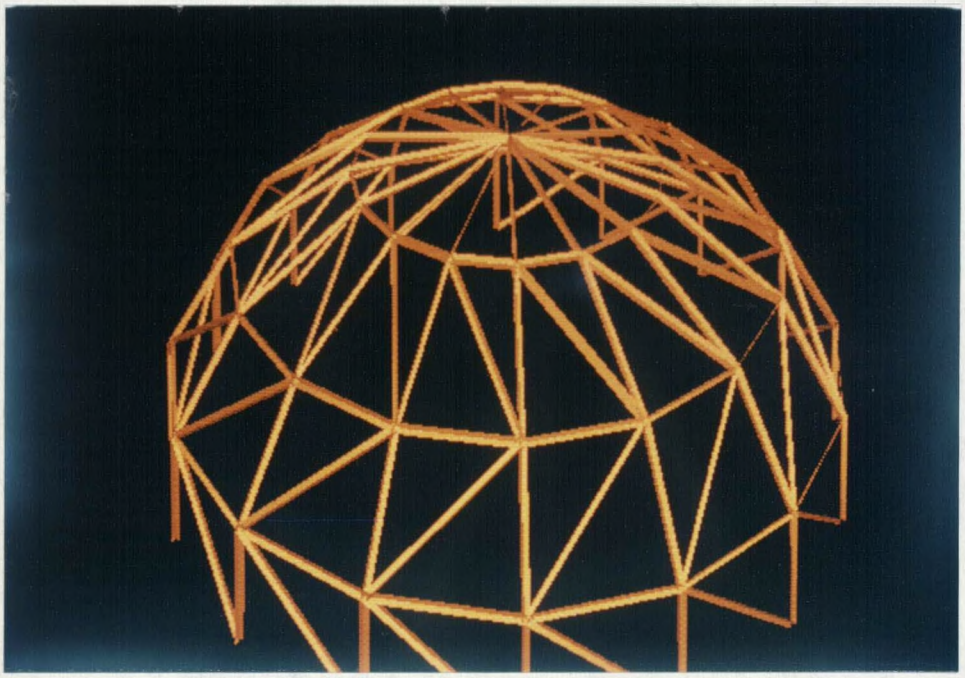


Figure 3.8: The steel semi-sphere structure

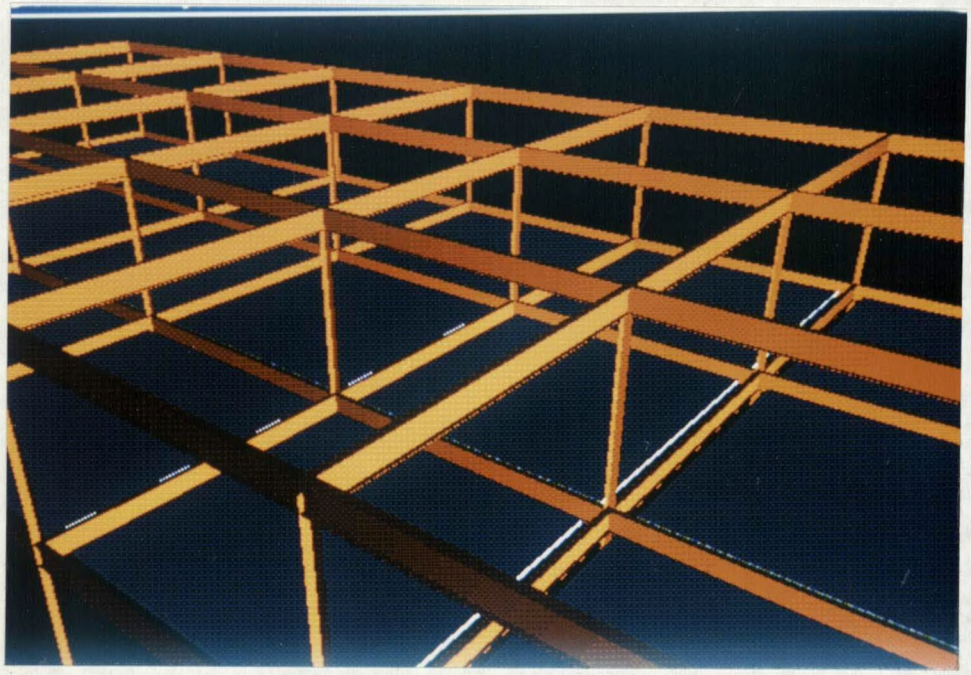


Figure 3.9: The steel and concrete structure (1)

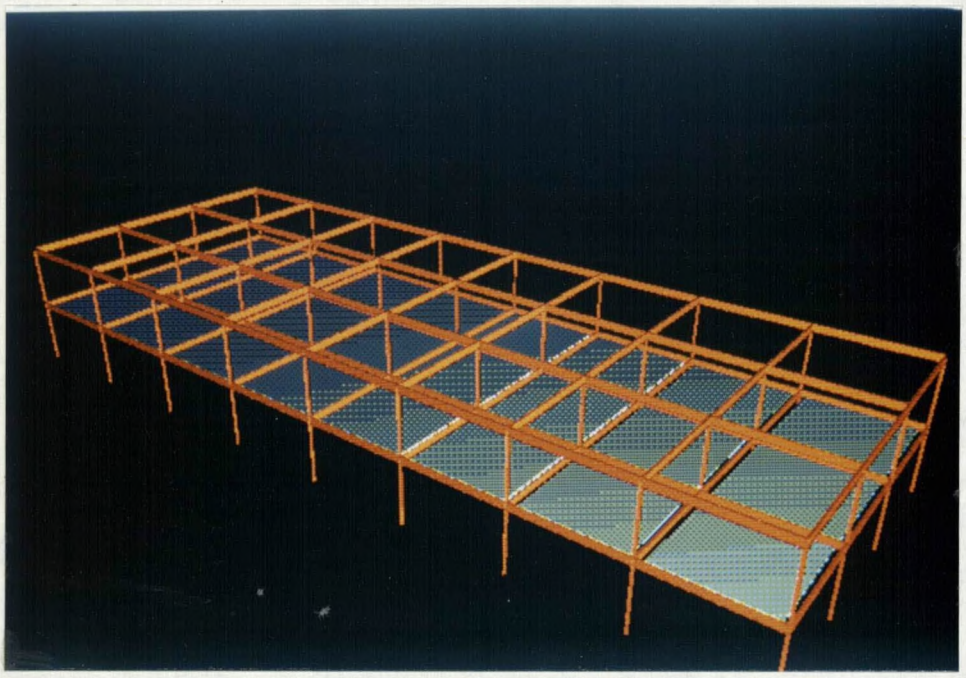


Figure 3.10: The steel and concrete structure (2)

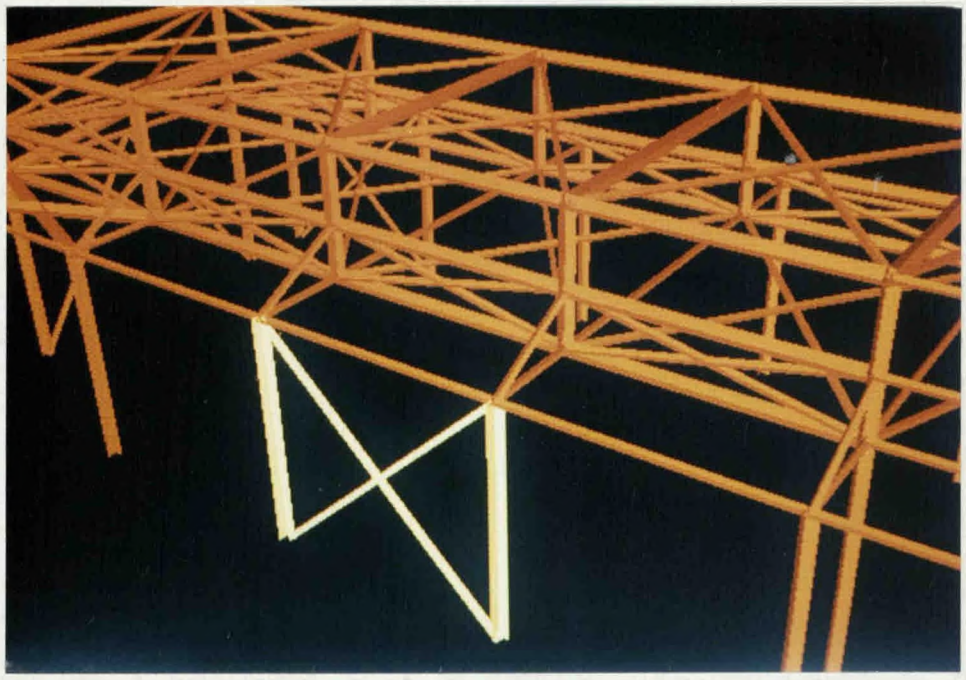


Figure 3.11: The steel and wood structure

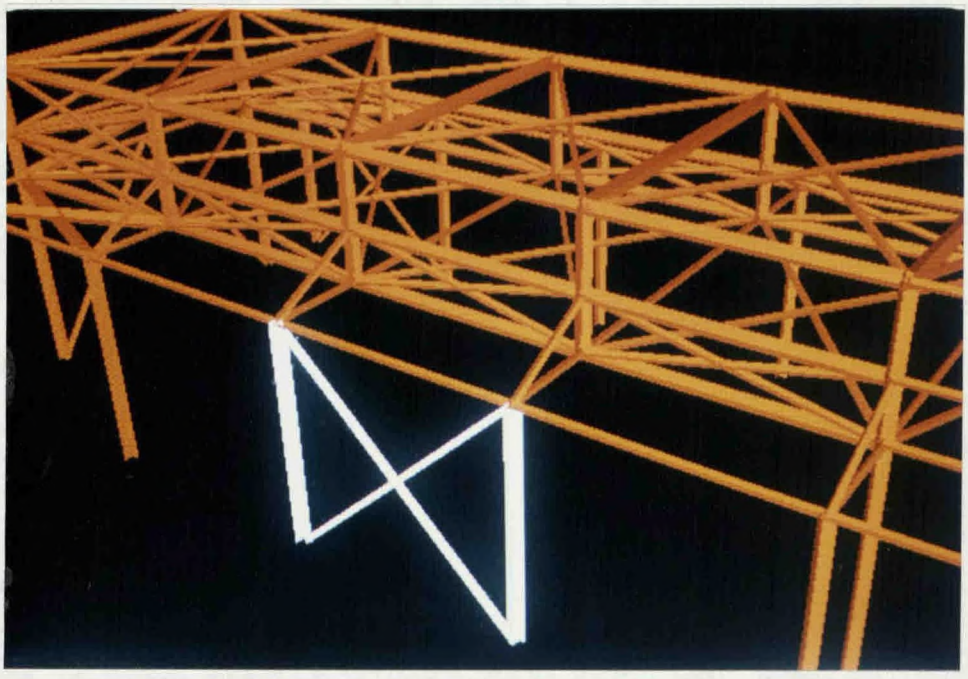


Figure 3.12: The steel and aluminum structure

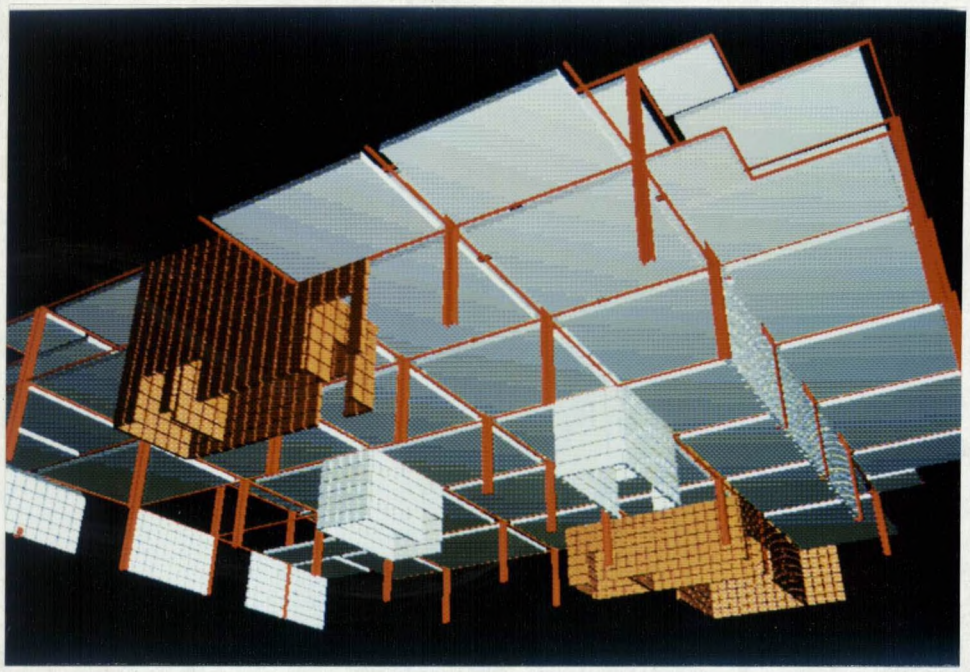


Figure 3.13: The steel, concrete and masonry structure

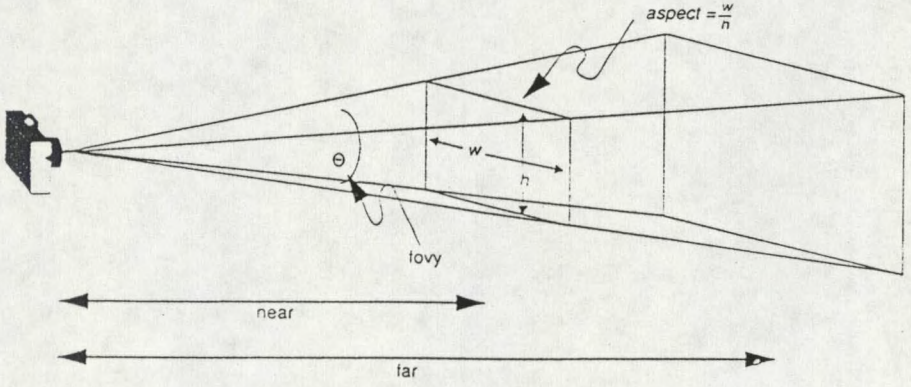


Figure 3.14: The perspective viewing volume specified by gluPerspective()

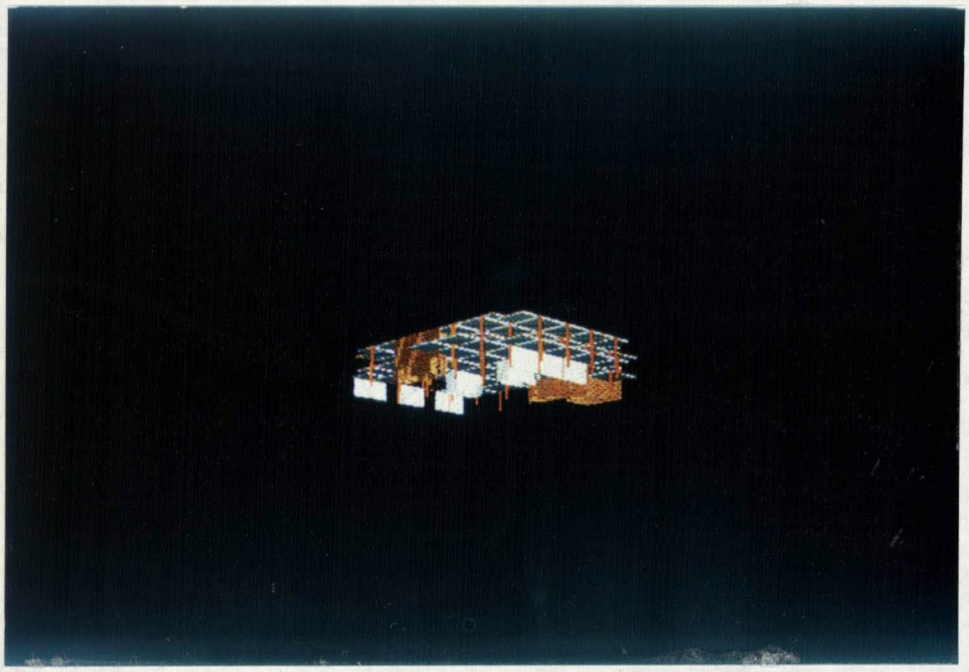


Figure 3.15: The perspective viewing effect: zoom in

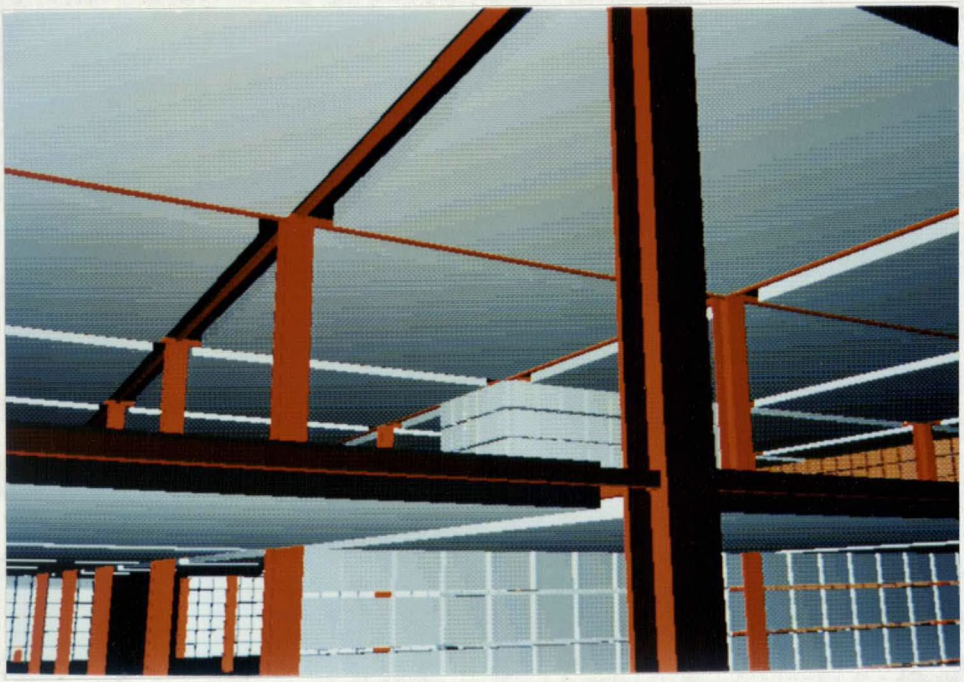


Figure 3.16: The perspective viewing effect: zoom out

Chapter 4

Conclusions and Discussion

Creating realistic images of 3D scenes is a complicated task because it involves a number of stages and various topics. In this paper, we discussed the most important techniques used for creating realistic images, such as coloring, lighting, shading and viewing, and presented an efficient implementation by integrating them together into the Visual Analysis software package. Finally, the performance of Visual Analysis has been greatly improved and the more realistic images can be rendered.

It should be indicated that although choosing the parameters and OpenGL commands can be based on the mathematical models discussed in Chapter 2, most of them are still empirical values but can create satisfactory effects (see Figures 3.1 to 3.16) for building design.

For creating more realistic images, we still should consider more techniques mentioned in Chapter 2, such as texture mapping, casting shadows, improving camera models and transparency. By using texture mapping, the wood or masonry (brick wall) can be more realistic because the texture mapping can mimic the surface details of real objects. Casting shadows can also improve the realistic effects for a building. Although it is not necessarily more desirable or useful in Visual Analysis, it can improve the aesthetic effects. OpenGL doesn't provide the simple commands for casting shadows. See [35] [36] for more details about casting shadows in OpenGL.

Also, the techniques of transparency or improving camera models are not used in Visual Analysis, but they are very useful for the photographic realism of pictures. They are also significant for further research. Another topic is lighting. We have discussed the various lighting models, but we didn't use area light in Visual Analysis because there are no simple OpenGL commands related to area light. If we build our own area light model and use it in Visual Analysis, it should enhance the realistic effects of images. See [37] [38] for more details about area light.

Due to the time limitation, the simplicity of OpenGL, and the hardware limitation (only on a PC, not an SGI workstation platform), these functions have not been implemented in Visual Analysis, but can be included in the future and can generate better effects for realistic images.

Bibliography

[1] Hagen, M., *Varieties of Realism*, Cambridge University Press, Cambridge, England, 1986.

[2] Saito, T., and Takahashi, T., "Comprehensible Rendering of 3-D Shapes", *Computer Graphics*, PP. 197-206, 24(4), August 1990.

[3] Haeberli, P., "Paint by Numbers: Abstract Image Representation", *Computer Graphics*, PP. 207-214, 24(4), August 1990.

[4] Hanrahan, P., and Haeberli, P., "Direct WYSINYG Painting and Texturing on 3D Shapes", *Computer Graphics*, PP. 215-223, 24(4), August 1990.

[5] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, Addison-Wesley Publishing Company, February 1994.

[6] Kaneda, K., Okamoto, T., Nakamae, E., and Nishita, T., "Highly Realistic Visual Simulation of Outdoor Scenes Under Various Atmospheric Conditions", *CG International '90*, PP. 177-131, 1990.

[7] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, Addison-Wesley Publishing Company, June 1995.

[8] Nishita, T., and Nakamae E., "Half-Tone Representation of 3-D Objects Illuminated by Area Sources or Polyhedron Sources", *IEEE COMPSAC*, PP. 237-242, 1983.

[9] Klassen R.V., "Modeling the Effect of the Atmosphere on Light", *ACM Trans. on Graphics*, PP. 215-237, 6(3), 1987.

[10] Inakage M., "An Illumination Model for Atmospheric Environment", *Proc. CGI'89*, PP. 533-548, 1989.

[11] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 410-413, Addison-Wesley Publishing Company, February 1994.

[12] O'Rourke, M. *Principles of Three-Dimensional Computer Animation.* PP. 79, W.W. Norton & Company, 1995.

[13] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 479, Addison-Wesley Publishing Company, February 1994.

[14] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 480, Addison-Wesley Publishing Company, February 1994.

[15] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 482, Addison-Wesley Publishing Company, February 1994.

[16] Phong, Bui-Tuong, "Illumination for Computer Generated Pictures", *CACM*, PP. 311-317, 18(6), June 1975.

[17] Hall, R., *Illumination and Color in Computer Generated Imagery*, Springer-Verlag, New York, 1989.

[18] Cok, R., and K. Torrance, "A Reflectance Model for Computer Graphics", *ACM TOG*, PP. 7-24, 1(1), January 1982.

[19] Kajiya, J.T., "Ray Tracing Parametric Patches", *GIGGRAPH '82*, PP. 245-254, 1982.

[20] Cabral, B., N. Max, and R. Springmeyer, "Bidirectional Reflection Functions from Surface Bump Maps", *SIGGRAPH '87*, PP. 273-281, 1987.

[21] Wolff, L., and D. Kurlander, "Ray Tracing with Polarization Parameters", *CG & A*, PP. 44-55, November 1990.

[22] He, X., P. Heynen, R.Phillips, K. Torrance, D.Salesin, and D.Greenberg, "A Fast and Accurate Light Reflection Model", *SIGGRAPH 92*, PP. 253-254, 1992.

[23] O'Rourke, M., *Pronciples of Three-Dimensional Computer Animation.* PP. 83, W.W. Norton & Company, 1995.

[24] Gouraud, H., "Continuous Shading of Curved Surfaces", *IEEE Trâns. on Computers*, PP. 623-629, C-20(6), June 1971.

[25] Phong, Bui-Tuong, "Illumination for Computer Generated Pictures", *CACM*, PP. 311-317, 18(6), June 1975.

[26] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 496-498, Addison-Wesley Publishing Company, February 1994.

[27] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 195, Addison-Wesley Publishing Company, February 1994.

[28] Zhou, C., *Based on Basic Graphic Principles Compare PEXlib 5.2 and OpenGL 1.0*, Montana State University, July 1996.

[29] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 234, Addison-Wesley Publishing Company, February 1994.

[30] Foley, J.D., Van Dam, A., Feiner, S.K., Hughes, J.F., and Phillips, R.L., *Introduction to Computer Graphics*, PP. 498-525, Addison-Wesley Publishing Company, February 1994.

[31] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, PP. 171, Addison-Wesley Publishing Company, June 1995.

[32] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, PP. 177, Addison-Wesley Publishing Company, June 1995.

[33] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, PP. 191-192, Addison-Wesley Publishing Company, June 1995.

[34] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, PP. 90-94, Addison-Wesley Publishing Company, June 1995.

[35] Harasta, J., "Introduction to Casting Shadows in OpenGL/Mesa", 1997.

[36] Neider, J., Davis, T., and Woo, M., *OpenGL Programming Guide*, PP. 90-94, Addison-Wesley Publishing Company, June 1995.

[37] Picott, K.P., "Extensions of the Linear and Area Lighting Models", *IEEE Computer Graphics and Applications*, PP. 31-38, March 1992.

[38] Lischinski, D., Tampieri, F., and Greenberg, D.P., "Discontinuity Meshing for Accurate Radiosity", *IEEE Computer Graphics and Applications*, PP. 25-39, November 1992.