UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Chapter 3

Computer Graphics Overview

3.1 Introduction

The calculation of computer generated images takes up a large amount of processor time. The field of computer graphics was developed to increase the speed and fidelity of computer generated images. The purpose of this chapter is to highlight some of the fundamentals of computer graphics that are relevant to this study. A short overview is given of the conventions used in computer graphics and matrix operations. Two and three-dimensional transforms are investigated, leading to a discussion on perspective and orthogonal projections. Texture maps are used to simplify the generation of physically realistic imagery. The main elements of the OpenGL API are discussed. This chapter is based on Foley *et al.* [15], except where otherwise noted.

3.2 History of computer graphics

A short history of computer graphics is given by Foley *et al.* [15].

> Plotting on hard-copy devices such as teletypes and line printers date from the earliest computers. The Whirlwind Computer, developed at MIT in 1950, used computer driven CRT (cathode ray tube) displays for output. The SAGE Air Defense System in the middle of the 1950s used CRT display consoles on which operators identified targets by pointing at them with light pens. Ivan Sutherland founded modern interactive computer graphics with the development of the Sketchpad drawing system as part of his Ph.D. work in 1963. He introduced data structures for storing symbol hierarchies which are built up by replication of standard components as well as interaction techniques for using the keyboard and a light-pen for choice-making, pointing and drawing. Computer graphics were then largely driven by developments in computer-aided design and computer-aided manufacturing. The display devices developed in the mid-sixties are called vector displays. They consist of a display processor, a display buffer memory and a CRT. The buffer stores the computer generated display-list or display program which contained point and line plotting commands as well as character plotting commands. These commands are interpreted by the display processor, which controls the display on the CRT. Vector displays of up to 4096x4096 points were

UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

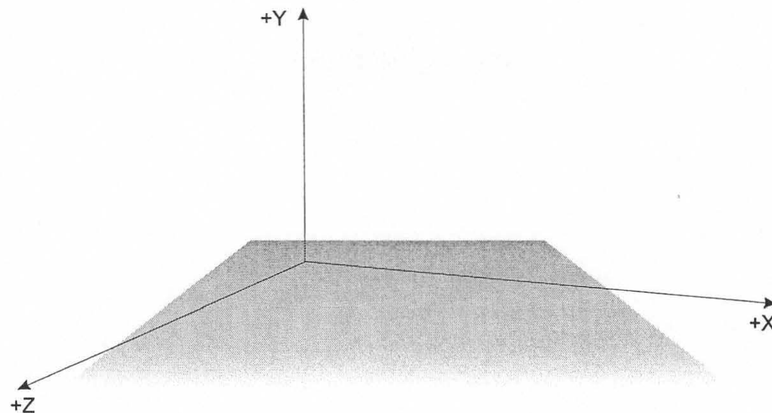

Figure 3.1: Right-handed coordinate system

developed.

In the seventies the generation of images was moved from the central computer to a mini-computer connected to the display device. The most significant development in the mid-seventies, however, was the appearance of low-cost solid state memory. This lead to the development of raster displays. In the raster displays, primitives such as lines, characters and polygons are stored in a refresh buffer in terms of their component points, called pixels. The image is formed from the raster. The raster is simply a matrix of pixels covering the display area.

OpenGL was developed from SGI's Iris GL [16]. Iris GL was used by SGI's high-end graphics workstations as a 3D API (application programming interface). Iris GL required specialised hardware to display graphics. OpenGL was developed to be a more portable graphics API than Iris GL. SGI made the OpenGL available to other vendors through licensing and the OpenGL ARB (architecture review board) was formed. The founding members of the OpenGL ARB were SGI, DEC, IBM, Intel and Microsoft. Version 1.0 of the OpenGL specification was introduced on July 1, 1992. The current specification of OpenGL, Version 1.2.1, was released on April 1, 1999.

3.3 Coordinate Convention

The OpenGL library uses a right-handed coordinate system as illustrated in Figure 3.1. This is from Wright *et al.* [16, p137].

3.4 Two-dimensional transformations

Matrix operations are fundamental to the implementation of a computer graphics system. Sections 3.4 and 3.5 are included to highlight the basic matrix transformations required for two- and three-dimensional computer graphics.

The multiplication of matrix $A = (a_{ij})$, a $m \times n$ matrix, and matrix $B = (b_{ij})$, a $n \times p$ matrix, is defined by de la Rosa *et al.*[17, p205] as the matrix $AB = (c_{ij})$ where

$$
\begin{aligned}
c_{ij} &= (a_{i1}, a_{i2}, \dots, a_{in}) \cdot (b_{1i}, b_{2i}, \dots, b_{3i}), \\
&= \sum_{t=1}^{n} a_{it} b_{tj}, \quad (1 \leq i \leq m, \quad 1 \leq j \leq p). \qquad (3.1)
\end{aligned}
$$

AB is an $m \times p$ matrix.

The discussion on two-dimensional transformations is based on Foley *et al.*[15]. Points in the xy-plane can be translated to new positions by adding translation amounts to the points. If the point is to be moved kx units parallel to the x-axis and ly units parallel to the y-axis, the coordinates of the new position is given by:

$$
\begin{aligned}
x' &= x + kx, \qquad (3.2) \\
y' &= y + ly. \qquad (3.3)
\end{aligned}
$$

Similarly, an object can be translated by applying Equation (3.2) to each point of the object. All points on a line can be translated by translating the endpoints and drawing a line between the translated endpoints.

The points (as end-points of vectors) can also be scaled along the x-axis and y-axis by multiplying by S_x and S_y respectively. The new endpoints are given by:

$$
\begin{aligned}
x' &= S_x \times x, \qquad (3.4) \\
y' &= S_y \times y. \qquad (3.5)
\end{aligned}
$$

In the case of differential scaling the values of S_x and S_y are not equal, leading to a change in the proportions of an object made up of multiple points.

The points can also be rotated through an angle θ about the origin. The coordinates of the new point are given by:

$$
\begin{aligned}
x' &= x \times \cos\theta - y \times \sin\theta, \qquad (3.6) \\
y' &= x \times \sin\theta + y \times \cos\theta. \qquad (3.7)
\end{aligned}
$$

Equations (3.6) and (3.7) are given in matrix form by:

$$
[x' \quad y'] = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix}. \qquad (3.8)
$$

The translation, scaling and rotation of an object around the origin is illustrated in Figure 3.2.

The matrix representations for translation, scaling and rotation are:

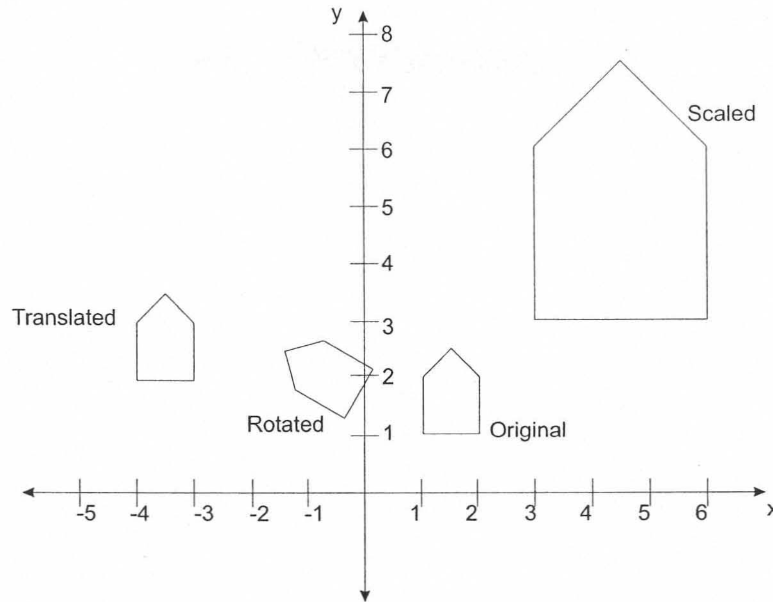

Figure 3.2: The translation, scaling and rotation of an object

$$\begin{aligned} P' &= P + T \quad \text{Translation,} && (3.9) \\ P' &= P \times S \quad \text{Scaling,} && (3.10) \\ P' &= P \times R \quad \text{Rotation.} && (3.11) \end{aligned}$$

Translation is the result of an addition, whereas scaling and rotation are the results of matrix multiplications. Ideally, all three transformations should be treated in a similar way, so that they can be combined in a single matrix operation. If the points are expressed in *homogeneous coordinates*, all three transformations can be treated as multiplications. In homogeneous coordinates, point $P(x, y)$ is represented as $P(W \times x, W \times y, W)$ for any scale factor $W \neq 0$. Given a homogeneous coordinate representation of a point $P(X, Y, W)$, the two-dimensional cartesian representation of the point is $x = \frac{X}{W}$ and $y = \frac{Y}{W}$. Points are now 3-element row vectors, leading to 3×3 transformation matrices to obtain another 3-element row vector. The translation matrix is now given by:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1] \times \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{bmatrix}, \qquad (3.12)$$

with D_x and D_y the translation in x and y, respectively.
The scaling matrix is given by:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1] \times \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix}, \qquad (3.13)$$

with S_x and S_y the scaling in the x and y dimensions, respectively.

Table 3.1: Scaled, rotated and translated coordinate points

Original x	Original y	Scaled x	Scaled y	Rotated x	Rotated y	Translated x	Translated y
1.000	1.000	3.000	3.000	-0.366	1.366	-3.000	-2.000
2.000	1.000	6.000	3.000	0.134	2.232	-2.000	2.000
2.000	2.000	6.000	6.000	-0.732	2.732	-2.000	3.000
1.500	2.500	4.500	7.500	-1.415	2.549	-2.500	3.500
1.000	2.000	3.000	6.000	-1.232	1.866	-3.000	3.000

The rotation matrix is given by:

$$[x' \quad y' \quad 1] = [x \quad y \quad 1] \times \begin{bmatrix} \cos\theta & \sin\theta & 0 \\ -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}, \tag{3.14}$$

with θ the rotation around the origin. The effects of the different transformations are illustrated in Figure 3.2. The original coordinates of the polygon are given with the scaled, rotated and translated coordinates in Table 3.1. The polygon is scaled by *(2,2)*, rotated +60°around the origin and translated by *(-4,1)*, respectively.

3.5 Three-dimensional transformations

The discussion on three-dimensional transformations is based on Angel [18]. Points in the three-dimensional space can also be translated, rotated and scaled around a reference point. Translation moves a point $\mathbf{p}$ to a new point $\mathbf{p}'$ using a displacement vector $\mathbf{d}$. This can be written as:

$$\mathbf{p} = [x \quad y \quad z], \quad \text{(the original point)} \tag{3.15}$$

$$\mathbf{d} = [\alpha_x \quad \alpha_y \quad \alpha_z], \quad \text{(the translation vector)} \tag{3.16}$$

$$\mathbf{p}' = \mathbf{p} + \mathbf{d}, \quad \text{(the new point)} \tag{3.17}$$

therefore

$$x' = x + \alpha_x, \tag{3.18}$$

$$y' = y + \alpha_y, \tag{3.19}$$

$$z' = z + \alpha_z. \tag{3.20}$$

In homogeneous coordinates this is given as:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \times \begin{bmatrix} 1 & 0 & 0 & \alpha_x \\ 0 & 1 & 0 & \alpha_y \\ 0 & 0 & 1 & \alpha_z \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.21}$$

i 16035665
b15433055

In the case of scaling and rotating, there is a fixed point that is unchanged by the transformation. For the cases where the fixed point is not the origin, transformations can be concatenated to obtain the transformation for the point. A point can be scaled, with the fixed point at the origin, using:

$$x' = \beta_x x, \tag{3.22}$$
$$y' = \beta_y y, \tag{3.23}$$
$$z' = \beta_z z. \tag{3.24}$$

These equations can be combined in homogeneous form as:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \times \begin{bmatrix} \beta_x & 0 & 0 & 0 \\ 0 & \beta_y & 0 & 0 \\ 0 & 0 & \beta_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.25}$$

In rotating a point around the origin, there are three possible degrees of freedom, corresponding to the ability to rotate the point independently about the three coordinate axis. The sequence of rotation is important because a rotation about the x axis by an angle θ followed by a rotation about the y axis by an angle ϕ would not give the same results as when the order of rotations were reversed. The equations for three-dimensional rotation about an axis can be derived by noticing that, for example, rotating around the z axis is equivalent to rotating in two dimensions in the xy plane, while keeping z constant. From Equation (3.14) the equations for rotation about the z axis by an angle of θ is given by:

$$x' = x \cos\theta - y \sin\theta, \tag{3.26}$$
$$y' = x \sin\theta + y \cos\theta, \tag{3.27}$$
$$z' = z. \tag{3.28}$$

In matrix form, rotation about the z axis can be written as:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \times \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.29}$$

Rotation about the x axis is given by:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \times \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \tag{3.30}$$

whereas rotation about the y axis is given by:

$$[x' \quad y' \quad z' \quad 1] = [x \quad y \quad z \quad 1] \times \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{3.31}$$

Rotation of an object around a point $\mathbf{p_0}$ would start by a translation to the origin, rotating the object around the x, y and z axis as required and then translating the resulting object back to $\mathbf{p_0}$.

When it is required to do successive transformations on a point $\mathbf{p}$, the transformations can be concatenated. In Equation (3.32) three transformations, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are carried out on $\mathbf{p}$, resulting in $\mathbf{q}$:

$$\mathbf{q} = \mathbf{CBAp}, \tag{3.32}$$
$$= (\mathbf{C}(\mathbf{B}(\mathbf{Ap}))). \tag{3.33}$$

In the case of many points that must be transformed, a new transformation, $\mathbf{M}$, can be calculated, with

$$\mathbf{M} = \mathbf{CBA}. \tag{3.34}$$

The matrix in Equation (3.34) can be used to calculate $\mathbf{q}$, therefore

$$\mathbf{q} = \mathbf{Mp}. \tag{3.35}$$

The use of a transformation such as $\mathbf{M}$, can save a significant amount of calculations.

3.6 Orthogonal and perspective projections

Objects in a scene, such as shown in Figure 3.3, can be projected in two ways onto a display. The first is orthogonal projection as shown in Figure 3.4. In this case objects with the same dimensions appear the same size, regardless of their distance from the viewing plane. This type of projection is used in CAD drawings, architectural design and two-dimensional graphs.

The result of perspective projection is shown in Figure 3.5. This projection adds the effect that distant objects appear smaller than nearby objects. Perspective projection is widely used in simulation and three-dimensional animation and is responsible for a large part of the realism.

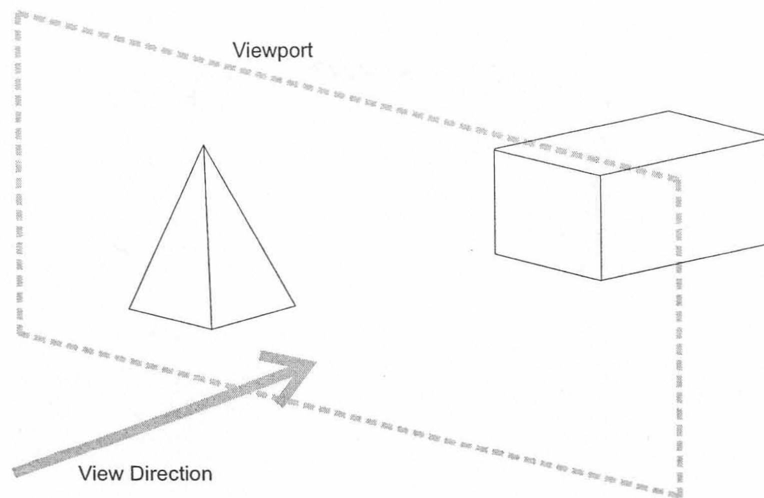

Figure 3.3: Terrain with two objects

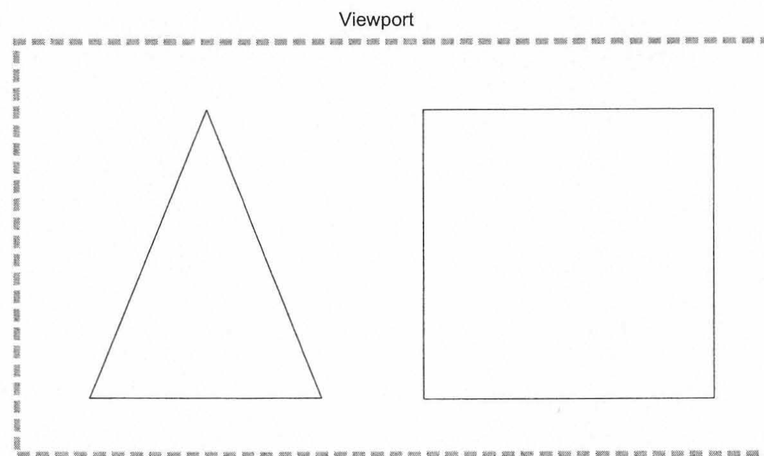

Figure 3.4: View using orthogonal projection

3.7 Texture maps

Wright *et al.* [16] regard texture mapping as one of the most significant developments in computer graphics in the past ten years. Texture mapping is used to supply realism to computer generated images. A texture map is an image that is fitted to polygons in a scene in order to generate realistic imagery. Texture mapping is used for applications that include anti-aliased text and reflection mapping. Texture mapping requires intensive calculations and can be slow on 2D graphics cards where all the calculations are done on the computer's CPU. Newer generation 3D graphics cards support texturing in hardware, leading to performance levels only found on graphics supercomputers a few years ago.

3.8 The open graphics library (OpenGL)

This section is based on Segal *et al.* [19]. OpenGL is concerned only with rendering into a framebuffer and reading values from the framebuffer. It does not provide support for devices such as mice and keyboards. OpenGL draws primitives subject to a number of selectable modes. Primitives include points, line segments, polygons and pixel rectangles. The primitives are defined by a group of one or more vertices. A vertex defines a point, an end-point of an edge, or a corner of a polygon where two edges meet. A vertex is defined by positional coordinates, colours, normals and texture coordinates.

The model for the interpretation of OpenGL commands is client-server. That means that a program (the client) issues commands and these commands are interpreted and processed by OpenGL (the server). The server and client may or may not be operating on the same computer. OpenGL is designed to run on a range of graphics platforms with varying capabilities and performance.

Commands are processed in the order in which they are received. A primitive must therefore be drawn completely before a subsequent one can influence the frame buffer. Another implication is that the results of pixel read operations are consistent when all previously invoked OpenGL commands are completed.

OpenGL commands are functions. Various groups of commands can perform the same operation, but differ in how arguments are supplied to them. The general syntax for a command declaration is:

rtype **Name**{ϵ**1234**}{ϵ **b s i f d ub us ui**}{ϵ**v**}
(*[args ,] T arg1,...,T argN [, args]*);

rtype is the return type of the function. The braces ({}) enclose a series of characters of which one is selected. ϵ indicates no character is selected. The arguments enclosed in brackets may or may not be present. If the final character is not **v**, then the number of arguments is **1**, **2**, **3**

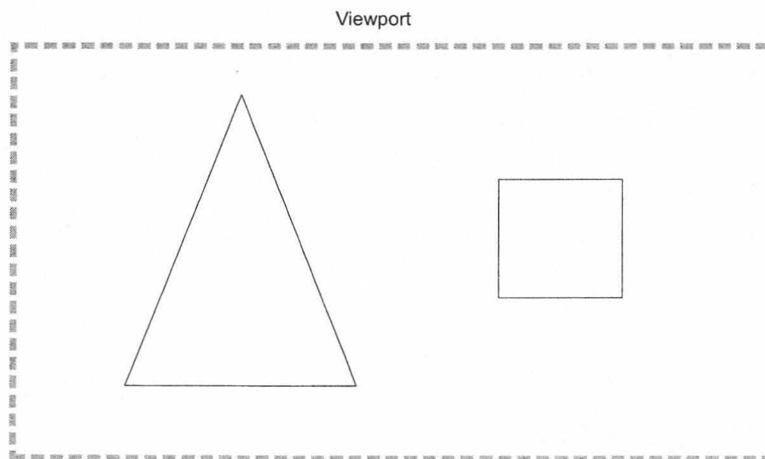

Figure 3.5: View using perspective projection

or **4**. If the final character is **v**, then only *arg1* is present and it is an array of N values of the indicated type.

The syntax can be illustrated using vertex specification as an example. Vertices are specified by giving their coordinates in two, three or four dimensions. The general versions of the command are:

> void **Vertex**{**234**}{**sifd**}(T *coords*) ;
> void **Vertex**{**234**}{**sifd**}**v**(T *coords*) ;

A call to any **Vertex** command specifies four coordinates: x, y, z and w. The x coordinate is the first coordinate, y the second, z the third and w the fourth. A call to **Vertex2** sets the x and y coordinates while the z coordinate is implicitly set to zero and w is set to one. **Vertex3** sets the x, y and z coordinates to the provided values, while w is set to one. **Vertex4** sets all four coordinates, allowing the specification of a point in homogeneous coordinates. The type specifiers **s**, **i**, **f** and **d** represent the standard types short, integer, float or double.

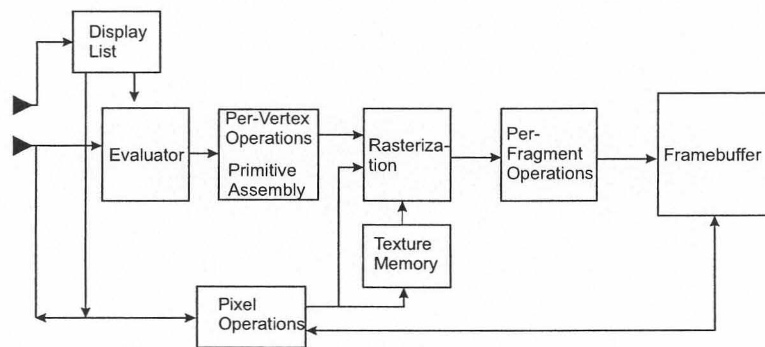

Figure 3.6: Block diagram of OpenGL

Figure 3.6 shows a schematic diagram of OpenGL. Some commands specify geometric objects to be drawn, whereas others control how the objects are handled by the various stages. Most commands may be accumulated in a display list for later processing. Commands that are not included in a display list are sent through a processing pipeline.

The first stage approximates curve and surface geometry by evaluating polynomial functions of input values. The next stage operates on geometric primitives described by vertices: points, line segments and polygons. In this stage vertices are transformed and lit and primitives are clipped to a viewing volume in preparation for the next stage. The rasterizer produces a series of framebuffer addresses and values using a two-dimensional description of a point, line segment or polygon. Each fragment produced by the rasterizer is fed to the next stage that performs operations on individual fragments before they finally alter the framebuffer. These operations include conditional updates into the framebuffer based on incoming and previously stored depth values, blending of incoming fragment colours with stored colours as well as masking and other logical operations on fragment values.

The vertex processing portion of the pipeline can be bypassed to send a block of fragments

directly to the individual fragment operations, eventually causing a block of pixels to be written to the framebuffer. Values may also be read back from the framebuffer or copied from one portion of the framebuffer to another.

Vertices, normals and texture coordinates are transformed before their coordinates are used to produce an image in the framebuffer. Figure 3.7 shows the sequence of transformations. The vertex coordinates that are presented to OpenGL are termed *object coordinates*. The *model-view matrix* is applied to these coordinates to yield *eye coordinates*. The next matrix, the projection matrix, is applied to the eye coordinates to yield *clip coordinates*. A perspective division is carried out on clip coordinates to yield *normalised device coordinates*. A final *viewport* transformation is applied to convert these coordinates into *window coordinates*.

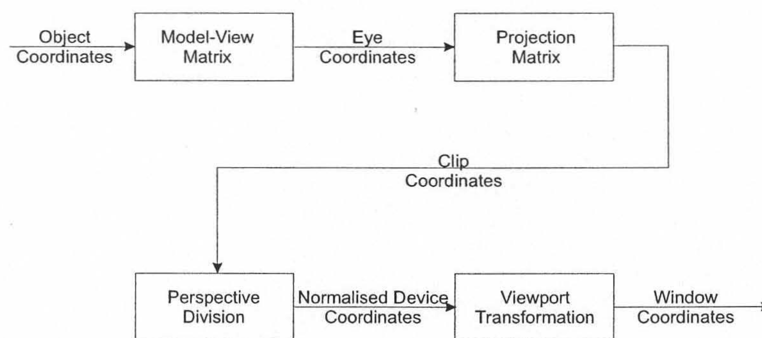

Figure 3.7: Vertex transformation sequence

3.9 OpenGL Utilities: GLUT

OpenGL was developed to be platform independent. It does not contain functions or commands for actions such as window management, keyboard input or mouse interaction. OpenGL was initially released with AUX, the OpenGL auxiliary library. The library was created to facilitate the learning and writing of OpenGL programs and did not contain basic GUI features.

AUX has been largely replaced by GLUT, the *OpenGL utility toolkit*. GLUT was written by Mark Kilgard at SGI. It includes pop-up menus, window management, keyboard and mouse interaction and even joy-stick support. GLUT makes it possible to port programs with relatively little effort between different platforms.

3.10 A short OpenGL program

This section shows a short OpenGL program that was written using GLUT. The listing is based on code from Wright *et al.*[16].

```
// A short OpenGL Program using GLUT

#include <windows.h>
```

```
#include <gl/glut.h>

// Main program entry point
void main(void)
    {
    // Set up a double-buffered display and specify
    // the pixel mode to be red, green and blue.
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    // Creates the display window with title "Short Program"
    glutCreateWindow("Short Program");
    // The function called by glutMainLoop whenever a new
    // scene is to be rendered.
    glutDisplayFunc(RenderScene);
    // Setup the rendering context.  Parameters such as the
    // background colour, lighting and fog can be set up in
    // this function.
    SetupRC();
    // The glutMainLoop calls functions to handle events such
    // as keyboard inputs, timer ticks and scaling of windows.
    glutMainLoop();
    }
void RenderScene(void)
    {
    // Clear the window with current clear color
    glClear(GL_COLOR_BUFFER_BIT);
    // Draw a filled triangle with a dark gray color
    glBegin(GL_TRIANGLES);
        glColor3ub(150, 150, 150);
        glVertex2f(-0.5, -0.5);
        glVertex2f(0.5, -0.5);
        glVertex2f(0.0, 0.5);
    glEnd();
    // Flush drawing commands
    glutSwapBuffers();
    }
// Setup the rendering state
void SetupRC(void)
    {
    // Set clear color to light gray
    glClearColor(0.90f, 0.90f, 0.90f, 1.0f);
    }
```

The output of the program is shown in Figure 3.8.

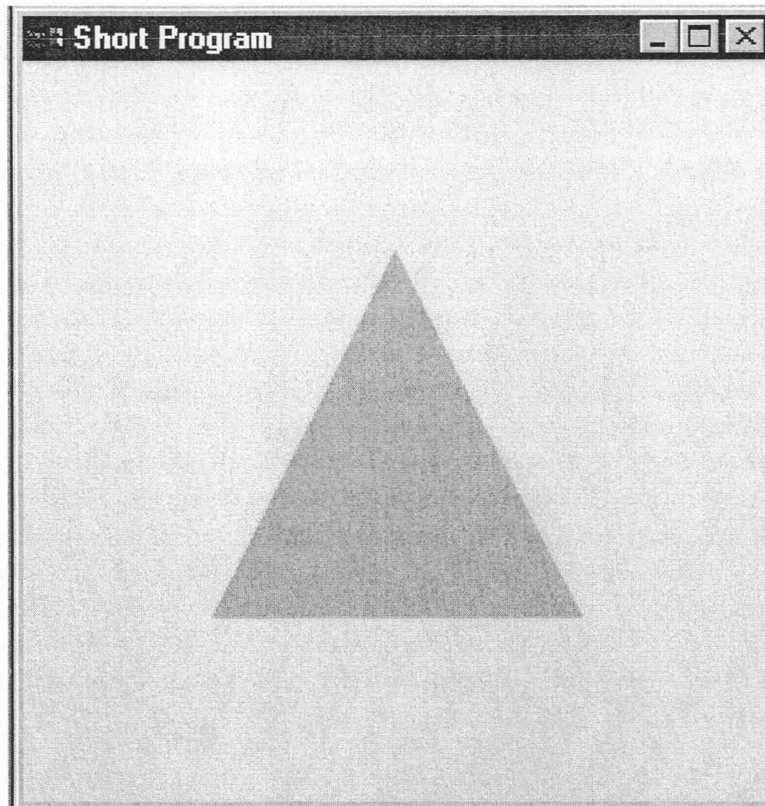

Figure 3.8: Image generated by using the short GLUT and OpenGL program

3.11 The use of OpenGL commands to simulate an infrared scenario

3.11.1 glColor

A colour can be defined for each vertex. If an infrared scenario is simulated, the colour assigned to a vertex can be replaced with a radiance value. The colours in OpenGL consist of a combination of red, green and blue. The radiance value can be assigned only to a single colour. It is also possible to assign the same radiance value to all three colours. This would lead to gray-scale images, which are similar to the output of typical thermal imagers. The "colour" output seen on some thermal imagers is a false colour map that is assigned according to the irradiance level of the pixel.

The commercially available graphics accelerators for the PC are limited to a resolution of eight bits per colour. It is therefore only possible to assign 256 radiance levels to an image. This can lead to severe reductions in the radiometric accuracy of the rendered images. A technique to artificially increase the resolution will be investigated in Chapter 4.

3.11.2 glFog

OpenGL provides depth cuing and atmospheric effects through the *glFog* function. Fog provides a way of adding a predefined colour to each vertex in the image, with the amplitude based on the distance between the vertex and the observer. OpenGL supports three kinds of fog namely:

GL_LINEAR: linear fog that is used for depth cuing.

GL_EXP: exponential fog that is used for heavy fog or clouds.

GL_EXP2: exponential fog that is used for smoke and weather haze.

Once a fog type is specified, the fog colour is specified using:

```
Glint fog_colour[4] = {red, green, blue, alpha};
glFogiv(GL_FOG_COLOR, fog_colour);
```

In addition to the fog colour, GL_EXP and GL_EXP2 have an additional density parameter:

```
glFogf(GL_FOG_DENSITY, fog_density);
```

The equivalent infrared atmospheric transmittance is defined using the *density* parameter, wheras the infrared path radiance is defined by specifying the *colour* of the fog. The effect of the three types of fog are shown in Figure 3.9. The data were generated using OpenGL with linear fog from 100m to 2000m, GL_EXP and GL_EXP2 fog with a density parameter of 0.001 and the fog colour set to $\{0, 0, 0\}$. The fog colour was black, resulting in no path radiance. The irradiance was normalised with the irradiance calculated for a scenario without fog.

Angel [18, p420] gives the equations for the different types of fog as:

$$\textbf{GL_LINEAR FOG}: \quad f = 1 - dz, \tag{3.36}$$

$$\textbf{GL_EXP FOG}: \quad f = e^{-dz}, \tag{3.37}$$

$$\textbf{GL_EXP2 FOG}: \quad f = e^{-(dz)^2}, \tag{3.38}$$

with d the density factor and z the distance. The linear fog is clamped between 1 and 0 using

```
glFogf(GL_FOG_START, start_of_fog);
glFogf(GL_FOG_END, end_of_fog);
```

with the transmittance 100% at the start of the fog and 0% at the end of the fog. Objects closer than GL_FOG_START are rendered without fog effects and objects further than GL_FOG_END are rendered with maximum fog effects. Equation (3.36) are modified by OpenGL to take this into account. The linear fog equation in this case becomes:

$$FOG = \begin{cases} 1 & \text{if z} < \text{GL_FOG_START}, \\ 1 - dz & \text{if GL_FOG_START} < \text{z} < \text{GL_FOG_END}, \\ 0 & \text{if z} > \text{GL_FOG_END}. \end{cases} \tag{3.39}$$

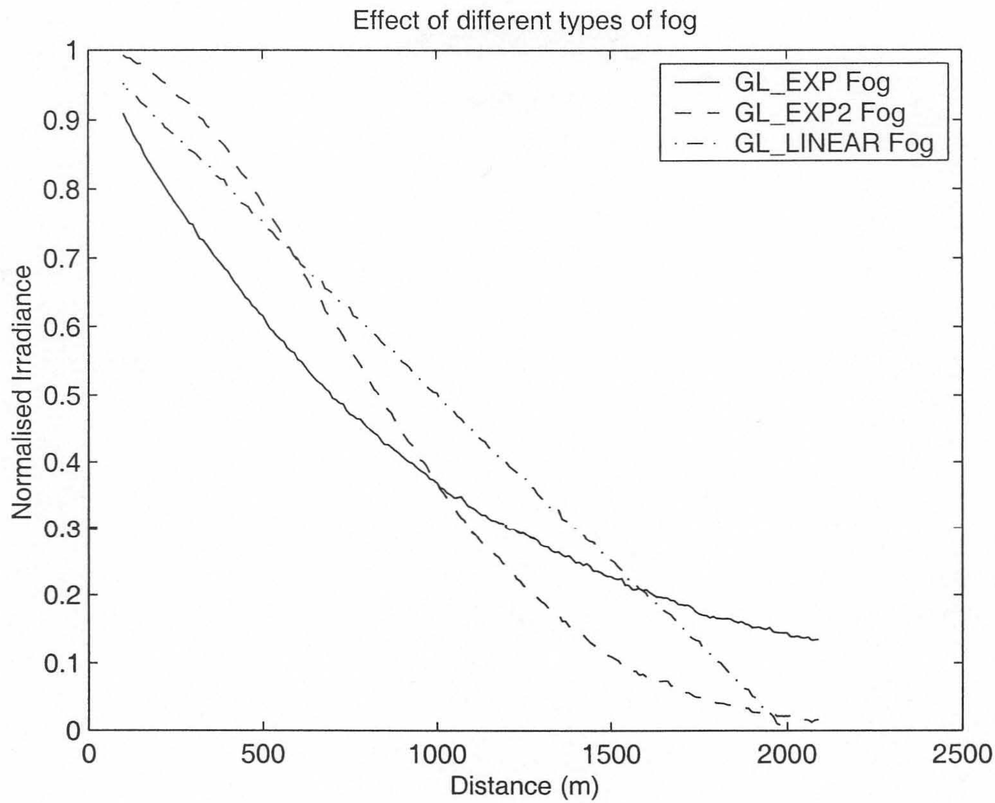

Figure 3.9: Demonstration of the reduction in normalised irradiance due to fog

3.11.3 The dynamic range of computer graphics hardware

The dynamic range of PC computer graphics hardware is limited to 8 bits per colour channel. It is not possible to increase the dynamic range in the hardware and another technique must be implemented to artificially increase the dynamic range. This technique is investigated in Chapter 4.