C++ is Fun – Part 12
at Turbine/Warner Bros.!

Russell Hanson

Syllabus

1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
3) Basic data structures and how to use them, opening files and performing operations on files – April 8-10
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
Project 1 Due – April 17
5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
7) Basic threads models and some simple databases SQLite May 6-8
8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
Project 2 Due May 15
9) How to download an API and learn how to use functions in that API, Windows Foundation Classes May 20-22
10) Designing and implementing a simple game in C++ May 27-29
11) Selected topics – Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC June 3-5
12) Working on student projects - June 10-12
Final project presentations Project 3/Final Project Due June 12

Woah! Tic Tac Toe!!!

```
Welcome to Tic-Tac-Toe
You will make your move by entering a number, 0-8.
The number will correspond to the board position as illustrated:
0 | 1 | 2
3 | 4 | 5
6 | 7 | 8

You want to go first? (y/n) : y
You want to be X or O? (x/o) : X
You want to be X or O? (x/o) : x
  |   |   |
  |   |   |
  |   |   |

Where do you want to move? (0 - 8): 0
X |   |   |
  |   |   |
  |   |   |

X |   |   |
  | O |   |
  |   |   |

Where do you want to move? (0 - 8): 8
X |   |   |
  | O |   |
  |   | X |

X |   | O |
  | O |   |
  |   | X |

Where do you want to move? (0 - 8): 6
X |   | O |
  | O |   |
X |   | X |

X |   | O |
  | O |   |
X | O | X |

Where do you want to move? (0 - 8): 1
X | X | O |
  | O |   |
X | O | X |

X | X | O |
O | O |   |
X | O | X |

Where do you want to move? (0 - 8): 5
X | X | O |
O | O | X |
X | O | X |


TIE!
sh: PAUSE: command not found
Russells-MacBook-Pro:mgraessle_Homework4 russell$
```

```
/*
Use the current empty pieces to have the computer
choose the best move to counter the last human move
*/
void Board::computerMove()
{
	// set the list of all the ways to win
	int WAYS_TO_WIN [8][3] = {{0, 1, 2},
										{3, 4, 5},
										{6, 7, 8},
										{0, 3, 6},
										{1, 4, 7},
										{2, 5, 8},
										{0, 4, 8},
										{2, 4, 6}};

	// if computer can win, take that move
	for (int i = 0; i < 8; ++i)
	{
		if ( board_pieces[WAYS_TO_WIN[i][0]] == COMPUTER && board_pieces[WAYS_TO_WIN[i][1]] == COMPUTER &&
board_pieces[WAYS_TO_WIN[i][2]] == EMPTY) {
			setPiece(WAYS_TO_WIN[i][2], COMPUTER);
			return;
		}

		else {
			if ( board_pieces[WAYS_TO_WIN[i][1]] == COMPUTER && board_pieces[WAYS_TO_WIN[i][2]] ==
COMPUTER && board_pieces[WAYS_TO_WIN[i][0]] == EMPTY) {
				setPiece(WAYS_TO_WIN[i][0], COMPUTER);
				return;
			}

			else {
				if ( board_pieces[WAYS_TO_WIN[i][0]] == COMPUTER &&
board_pieces[WAYS_TO_WIN[i][2]] == COMPUTER && board_pieces[WAYS_TO_WIN[i][1]] == EMPTY )
				{
					setPiece(WAYS_TO_WIN[i][1], COMPUTER);
					return;
				}
			}
		}
	}

	// if human can win, block that move
	for (int i = 0; i < 8; ++i)
```

```
				}
			}
		}

		// if human can win, block that move
		for (int i = 0; i < 8; ++i)
		{
			if ( board_pieces[WAYS_TO_WIN[i][0]] == HUMAN && board_pieces[WAYS_TO_WIN[i][1]] == HUMAN &&
board_pieces[WAYS_TO_WIN[i][2]] == EMPTY) {
				setPiece(WAYS_TO_WIN[i][2], COMPUTER);
				return;
			}

			else {
				if ( board_pieces[WAYS_TO_WIN[i][1]] == HUMAN && board_pieces[WAYS_TO_WIN[i][2]] == HUMAN
&& board_pieces[WAYS_TO_WIN[i][0]] == EMPTY) {
					setPiece(WAYS_TO_WIN[i][0], COMPUTER);
					return;
				}

				else {
					if ( board_pieces[WAYS_TO_WIN[i][0]] == HUMAN &&
board_pieces[WAYS_TO_WIN[i][2]] == HUMAN && board_pieces[WAYS_TO_WIN[i][1]] == EMPTY )
					{
						setPiece(WAYS_TO_WIN[i][1], COMPUTER);
						return;
					}
				}
			}
		}

		int BEST_MOVES[10] = {4, 0, 2, 6, 8, 1, 3, 5, 7};

		// if one of the best squares is empty, take it
		for (int i = 0; i < 10; ++i)
		{
			if ( isMoveLegal(BEST_MOVES[i]) )
			{
				setPiece(BEST_MOVES[i], COMPUTER);
				break;
			}// end if
		}// end for

}// end computerMove
```

Graphics and OpenGL!

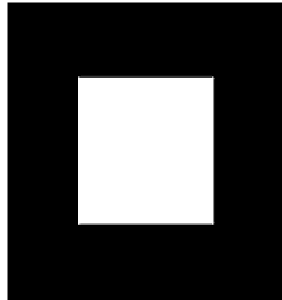


Figure 1-1 : White Rectangle on a Black Background

Example 1-1 : Chunk of OpenGL Code

```
#include <whateverYouNeed.h>

main() {

   InitializeAWindowPlease();

   glClearColor (0.0, 0.0, 0.0, 0.0);
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
   glBegin(GL_POLYGON);
      glVertex3f (0.25, 0.25, 0.0);
      glVertex3f (0.75, 0.25, 0.0);
      glVertex3f (0.75, 0.75, 0.0);
      glVertex3f (0.25, 0.75, 0.0);
   glEnd();
   glFlush();

   UpdateTheWindowAndCheckForEvents();
}
```

The first line of the **main**() routine initializes a *window* on the screen: The **InitializeAWindowPlease()** routine is meant as a placeholder for window system-specific routines, which are generally not OpenGL calls. The next two lines are OpenGL commands that clear the window to black: **glClearColor()** establishes what color the window will be cleared to, and **glClear()** actually clears the window. Once the clearing color is set, the window is cleared to that color whenever **glClear()** is called. This clearing color can be changed with another call to **glClearColor()**. Similarly, the **glColor3f()** command establishes what color to use for drawing objects - in this case, the color is white. All objects drawn after this point use this color, until it's changed with another call to set the color.

The next OpenGL command used in the program, **glOrtho()**, specifies the coordinate system OpenGL assumes as it draws the final image and how the image gets mapped to the screen. The next calls, which are bracketed by **glBegin()** and **glEnd()**, define the object to be drawn - in this example, a polygon with four vertices. The polygon's "corners" are defined by the **glVertex3f()** commands. As you might be able to guess from the arguments, which are (*x, y, z*) coordinates, the polygon is a rectangle on the z=0 plane.

OpenGL Command Syntax

As you might have observed from the simple program in the previous section, OpenGL commands use the prefix **gl** and initial capital letters for each word making up the command name (recall **glClearColor()**, for example). Similarly, OpenGL defined constants begin with GL_, use all capital letters, and use underscores to separate words (like GL_COLOR_BUFFER_BIT).

You might also have noticed some seemingly extraneous letters appended to some command names (for example, the **3f** in **glColor3f() and glVertex3f()**). It's true that the **Color** part of the command name **glColor3f()** is enough to define the command as one that sets the current color. However, more than one such command has been defined so that you can use different types of arguments. In particular, the **3** part of the suffix indicates that three arguments are given; another version of the **Color** command takes four arguments. The **f** part of the suffix indicates that the arguments are floating-point numbers. Having different formats allows OpenGL to accept the user's data in his or her own data format.

Some OpenGL commands accept as many as 8 different data types for their arguments. The letters used as suffixes to specify these data types for ISO C implementations of OpenGL are shown in Table 1-1, along with the corresponding OpenGL type definitions. The particular implementation of OpenGL that you're using might not follow this scheme exactly; an implementation in C++ or Ada, for example, wouldn't need to.

Suffix	Data Type	Typical Corresponding C-Language Type	OpenGL Type Definition
b	8-bit integer	signed char	GLbyte
s	16-bit integer	short	GLshort
i	32-bit integer	int or long	GLint, GLsizei
f	32-bit floating-point	float	GLfloat, GLclampf
d	64-bit floating-point	double	GLdouble, GLclampd
ub	8-bit unsigned integer	unsigned char	GLubyte, GLboolean
us	16-bit unsigned integer	unsigned short	GLushort
ui	32-bit unsigned integer	unsigned int or unsigned long	GLuint, GLenum, GLbitfield

Table 1-1 : Command Suffixes and Argument Data Types

OpenGL as a State Machine

OpenGL is a state machine. You put it into various states (or modes) that then remain in effect until you change them. As you've already seen, the current color is a state variable. You can set the current color to white, red, or any other color, and thereafter every object is drawn with that color until you set the current color to something else. The current color is only one of many state variables that OpenGL maintains. Others control such things as the current viewing and projection transformations, line and polygon stipple patterns, polygon drawing modes, pixel-packing conventions, positions and characteristics of lights, and material properties of the objects being drawn. Many state variables refer to modes that are enabled or disabled with the command **glEnable()** or **glDisable()**.

Each state variable or mode has a default value, and at any point you can query the system for each variable's current value. Typically, you use one of the six following commands to do this: **glGetBooleanv()**, **glGetDoublev()**, **glGetFloatv()**, **glGetIntegerv()**, **glGetPointerv()**, or **glIsEnabled()**. Which of these commands you select depends on what data type you want the answer to be given in. Some state variables have a more specific query command (such as **glGetLight*()**, **glGetError()**, or **glGetPolygonStipple()**). In addition, you can save a collection of state variables on an attribute stack with **glPushAttrib()** or **glPushClientAttrib()**, temporarily modify them, and later restore the values with **glPopAttrib()** or **glPopClientAttrib()**. For temporary state changes, you should use these commands rather than any of the query commands, since they're likely to be more efficient.

See Appendix B for the complete list of state variables you can query. For each variable, the appendix also lists a suggested **glGet*()** command that returns the variable's value, the attribute class to which it belongs, and the variable's default value.

OpenGL Rendering Pipeline

Most implementations of OpenGL have a similar order of operations, a series of processing stages called the OpenGL rendering pipeline. This ordering, as shown in Figure 1-2, is not a strict rule of how OpenGL is implemented but provides a reliable guide for predicting what OpenGL will do.

If you are new to three-dimensional graphics, the upcoming description may seem like drinking water out of a fire hose. You can skim this now, but come back to Figure 1-2 as you go through each chapter in this book.

The following diagram shows the Henry Ford assembly line approach, which OpenGL takes to processing data. Geometric data (vertices, lines, and polygons) follow the path through the row of boxes

that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) are treated differently for part of the process. Both types of data undergo the same final steps (rasterization and per-fragment operations) before the final pixel data is written into the framebuffer.

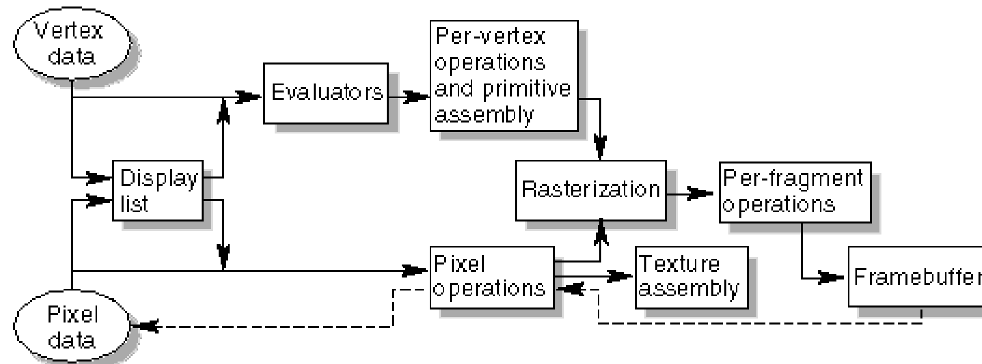


Figure 1-2 : Order of Operations

Display Lists

All data, whether it describes geometry or pixels, can be saved in a *display list* for current or later use. (The alternative to retaining data in a display list is processing the data immediately - also known as *immediate mode*.) When a display list is executed, the retained data is sent from the display list just as if it were sent by the application in immediate mode. (See Chapter 7 for more information about display lists.)

Evaluators

All geometric primitives are eventually described by vertices. Parametric curves and surfaces may be initially described by control points and polynomial functions called basis functions. Evaluators provide a method to derive the vertices used to represent the surface from the control points. The method is a polynomial mapping, which can produce surface normal, texture coordinates, colors, and spatial coordinate values from the control points. (See Chapter 12 to learn more about evaluators.)

Per-Vertex Operations

For vertex data, next is the "per-vertex operations" stage, which converts the vertices into primitives. Some vertex data (for example, spatial coordinates) are transformed by 4 x 4 floating-point matrices. Spatial coordinates are projected from a position in the 3D world to a position on your screen. (See Chapter 3 for details about the transformation matrices.)

If advanced features are enabled, this stage is even busier. If texturing is used, texture coordinates may be generated and transformed here. If lighting is enabled, the lighting calculations are performed using the transformed vertex, surface normal, light source position, material properties, and other lighting

Primitive Assembly

Clipping, a major part of primitive assembly, is the elimination of portions of geometry which fall outside a half-space, defined by a plane. Point clipping simply passes or rejects vertices; line or polygon clipping can add additional vertices depending upon how the line or polygon is clipped.

In some cases, this is followed by perspective division, which makes distant geometric objects appear smaller than closer objects. Then viewport and depth (z coordinate) operations are applied. If culling is enabled and the primitive is a polygon, it then may be rejected by a culling test. Depending upon the polygon mode, a polygon may be drawn as points or lines. (See "Polygon Details" in Chapter 2.)

The results of this stage are complete geometric primitives, which are the transformed and clipped vertices with related color, depth, and sometimes texture-coordinate values and guidelines for the rasterization step.

Pixel Operations

While geometric data takes one path through the OpenGL rendering pipeline, pixel data takes a different route. Pixels from an array in system memory are first unpacked from one of a variety of formats into the proper number of components. Next the data is scaled, biased, and processed by a pixel map. The results are clamped and then either written into texture memory or sent to the rasterization step. (See "Imaging Pipeline" in Chapter 8.)

If pixel data is read from the frame buffer, pixel-transfer operations (scale, bias, mapping, and clamping) are performed. Then these results are packed into an appropriate format and returned to an array in system memory.

There are special pixel copy operations to copy data in the framebuffer to other parts of the framebuffer or to the texture memory. A single pass is made through the pixel transfer operations before the data is written to the texture memory or back to the framebuffer.

Texture Assembly

An OpenGL application may wish to apply texture images onto geometric objects to make them look more realistic. If several texture images are used, it's wise to put them into texture objects so that you can easily switch among them.

Some OpenGL implementations may have special resources to accelerate texture performance. There may be specialized, high-performance texture memory. If this memory is available, the texture objects may be prioritized to control the use of this limited and valuable resource. (See Chapter 9.)

Rasterization

Rasterization is the conversion of both geometric and pixel data into *fragments*. Each fragment square corresponds to a pixel in the framebuffer. Line and polygon stipples, line width, point size, shading

Example 1-2 : Simple OpenGL Program Using GLUT: hello.c

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

void display(void)
{
/*  clear all pixels  */
    glClear (GL_COLOR_BUFFER_BIT);

/*  draw white polygon (rectangle) with corners at
 *  (0.25, 0.25, 0.0) and (0.75, 0.75, 0.0)
 */
    glColor3f (1.0, 1.0, 1.0);
    glBegin(GL_POLYGON);
        glVertex3f (0.25, 0.25, 0.0);
        glVertex3f (0.75, 0.25, 0.0);
        glVertex3f (0.75, 0.75, 0.0);
        glVertex3f (0.25, 0.75, 0.0);
    glEnd();

/*  don't wait!
 *  start processing buffered OpenGL routines
 */
    glFlush ();
}

void init (void)
{
/*  select clearing (background) color       */
    glClearColor (0.0, 0.0, 0.0, 0.0);

/*  initialize viewing values  */
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
}

/*
 *  Declare initial window size, position, and display mode
 *  (single buffer and RGBA).  Open window with "hello"
 *  in its title bar.  Call initialization routines.
 *  Register callback function to display graphics.
 *  Enter main loop and process events.
 */
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize (250, 250);
    glutInitWindowPosition (100, 100);
    glutCreateWindow ("hello");
    init ();
    glutDisplayFunc(display);

    glutMainLoop();
    return 0;   /* ISO C requires main to return int. */
}
```

Double Buffering

Most OpenGL implementations provide double-buffering - hardware or software that supplies two complete color buffers. One is displayed while the other is being drawn. When the drawing of a frame is complete, the two buffers are swapped, so the one that was being viewed is now used for drawing, and vice versa. This is like a movie projector with only two frames in a loop; while one is being projected on the screen, an artist is desperately erasing and redrawing the frame that's not visible. As long as the artist is quick enough, the viewer notices no difference between this setup and one where all the frames are already drawn and the projector is simply displaying them one after the other. With double-buffering, every frame is shown only when the drawing is complete; the viewer never sees a partially drawn frame.

If you are using the GLUT library, you'll want to call this routine:

void glutSwapBuffers(void);

Example 1-3 illustrates the use of **glutSwapBuffers()** in an example that draws a spinning square as shown in Figure 1-3. The following example also shows how to use GLUT to control an input device and turn on and off an idle function. In this example, the mouse buttons toggle the spinning on and off.

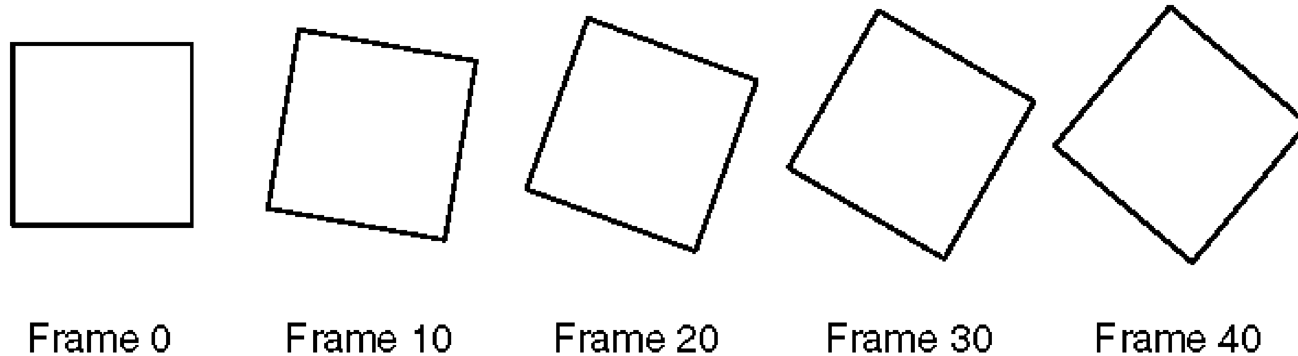


Figure 1-3 : Double-Buffered Rotating Square

Example 1-3 : Double-Buffered Program: double.c

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
#include <stdlib.h>

static GLfloat spin = 0.0;

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear(GL_COLOR_BUFFER_BIT);
   glPushMatrix();
   glRotatef(spin, 0.0, 0.0, 1.0);
   glColor3f(1.0, 1.0, 1.0);
```

```
   glRectf(-25.0, -25.0, 25.0, 25.0);
   glPopMatrix();
   glutSwapBuffers();
}

void spinDisplay(void)
{
   spin = spin + 2.0;
   if (spin > 360.0)
      spin = spin - 360.0;
   glutPostRedisplay();
}

void reshape(int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   glOrtho(-50.0, 50.0, -50.0, 50.0, -1.0, 1.0);
   glMatrixMode(GL_MODELVIEW);
   glLoadIdentity();
}

void mouse(int button, int state, int x, int y)
{
   switch (button) {
      case GLUT_LEFT_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(spinDisplay);
         break;
      case GLUT_MIDDLE_BUTTON:
         if (state == GLUT_DOWN)
            glutIdleFunc(NULL);
         break;
      default:
         break;
   }
}

/*
 *  Request double buffer display mode.
 *  Register mouse input callback functions
 */
int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
   glutInitWindowSize (250, 250);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMouseFunc(mouse);
   glutMainLoop();
   return 0;
}
```

Once a model's vertices have been clipped and transformed into window space, the GPU must determine what pixels in the viewport are covered by each graphics primitive. The process of filling in the horizontal spans of pixels belonging to a primitive is called *rasterization*. The GPU calculates the depth, interpolated vertex colors, and interpolated texture coordinates for each pixel. This information, combined with the location of the pixel itself, is called a *fragment*.

The process through which a graphics primitive is converted to a set of fragments is illustrated in Figure 0.4. An application may specify that *face culling* be performed as the first stage of this process. Face culling applies only to polygonal graphics primitives and removes either the polygons that are facing away from the camera or those that are facing toward the camera. Ordinarily, face culling is employed as an optimization that skips polygons facing away from the camera (*backfacing* polygons) since they correspond to the unseen far side of a model.

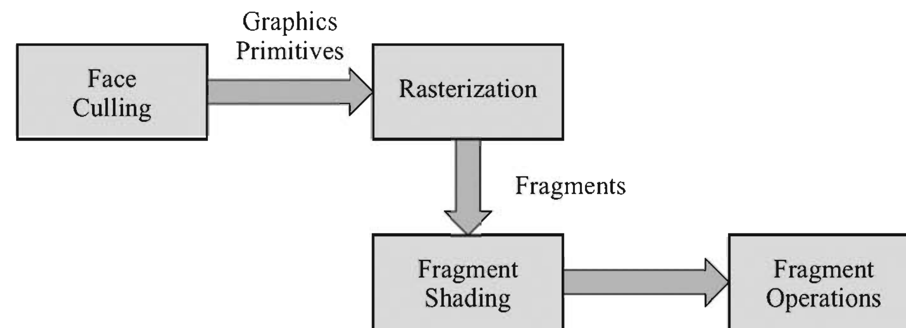


Figure 0.4 A graphics primitive is converted to a set of fragments during rasterization. After shading, fragments undergo the operations shown in Figure 0.5.

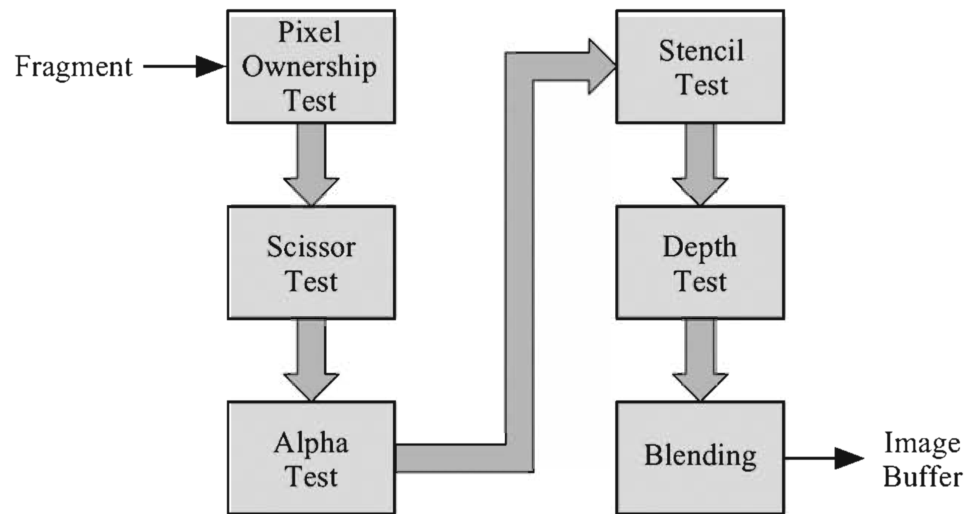


Figure 0.5 Operations performed before a fragment is written to the image buffer.

3.2 Scaling Transforms

To scale a vector **P** by a factor of a, we simply calculate $\mathbf{P}' = a\mathbf{P}$. In three dimensions, this operation can also be expressed as the matrix product

$$\mathbf{P}' = \begin{bmatrix} a & 0 & 0 \\ 0 & a & 0 \\ 0 & 0 & a \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}. \tag{3.9}$$

This is called a *uniform* scale. If we wish to scale a vector by different amounts along the x-, y-, and z-axes, as shown in Figure 3.1, then we can use a matrix that is similar to the uniform scale matrix, but whose diagonal entries are not necessarily all equal. This is called a *nonuniform* scale and can be expressed as the matrix product

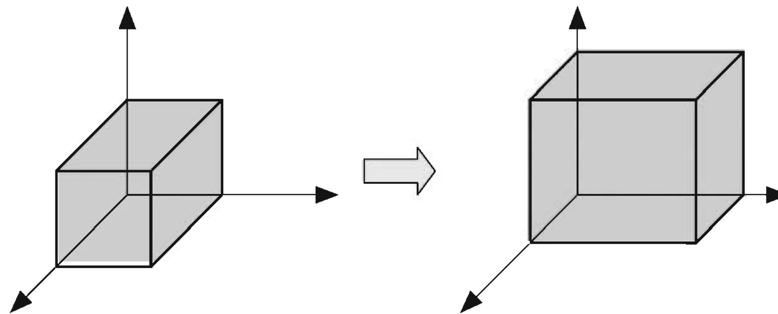


Figure 3.1 Nonuniform scaling.

$$\mathbf{P}' = \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix}. \tag{3.10}$$

A slightly more complex scaling operation that one may wish to perform is a nonuniform scale that is applied along three arbitrary axes. Suppose that we want to scale by a factor a along the axis $\mathbf{U}$, by a factor b along the axis $\mathbf{V}$, and by a factor c along the axis $\mathbf{W}$. Then we can transform from the $(\mathbf{U}, \mathbf{V}, \mathbf{W})$ coordinate system to the $(\mathbf{i}, \mathbf{j}, \mathbf{k})$ coordinate system, apply the scaling operation in this system using Equation (3.10), and then transform back into the $(\mathbf{U}, \mathbf{V}, \mathbf{W})$ coordinate system. This gives us the following matrix product.

$$\mathbf{P}' = \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix} \begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} U_x & V_x & W_x \\ U_y & V_y & W_y \\ U_z & V_z & W_z \end{bmatrix}^{-1} \begin{bmatrix} P_x \\ P_y \\ P_z \end{bmatrix} \tag{3.11}$$

3.3 Rotation Transforms

We can find 3×3 matrices that rotate a coordinate system through an angle θ about the x-, y-, or z-axis without much difficulty. We consider a rotation by a positive angle about the axis **A** to be that which performs a counterclockwise rotation when the axis **A** is pointing toward us.

First, we will find a general formula for rotations in two dimensions. As shown in Figure 3.2, we can perform a 90-degree counterclockwise rotation of a 2D vector **P** in the x-y plane by exchanging the x- and y-coordinates and negating the new x-coordinate. Calling the rotated vector **Q**, we have $\mathbf{Q} = \langle -P_y, P_x \rangle$. The vectors **P** and **Q** form an orthogonal basis for the x-y plane. We can therefore express any vector in the x-y plane as a linear combination of these two vectors. In particular, as shown in Figure 3.3, any 2D vector $\mathbf{P}'$ that results from the rotation of the vector **P** through an angle θ can be expressed in terms of its components that are parallel to **P** and **Q**. Basic trigonometry lets us write

$$\mathbf{P}' = \mathbf{P}\cos\theta + \mathbf{Q}\sin\theta. \quad (3.12)$$

This gives us the following expressions for the components of $\mathbf{P}'$.

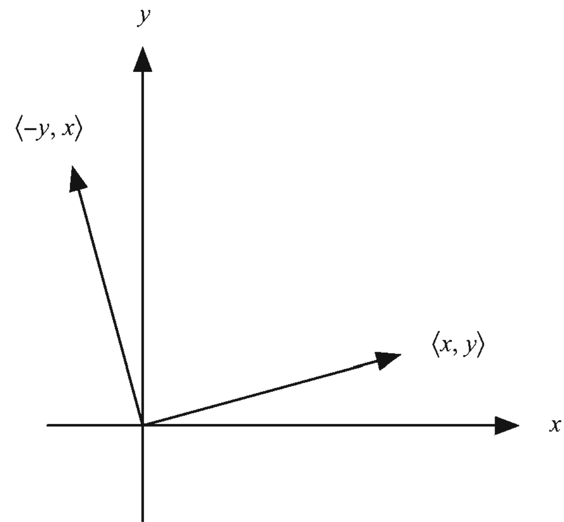


Figure 3.2 Rotation by 90 degrees in the *x*-*y* plane.

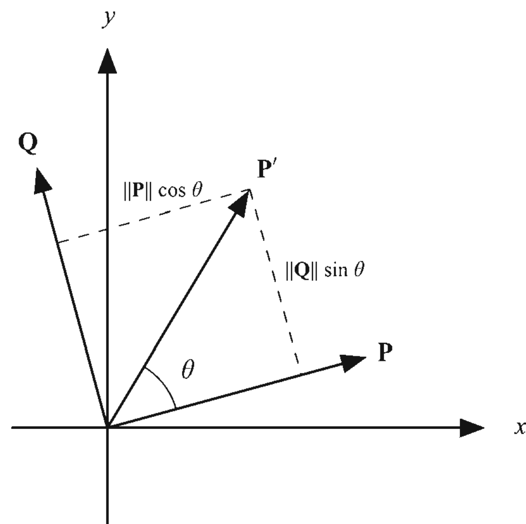


Figure 3.3. A rotated vector can be expressed as the linear combination of the original vector and the 90-degree counterclockwise rotation of the original vector.

$$\begin{aligned} P'_x &= P_x \cos\theta - P_y \sin\theta \\ P'_y &= P_y \cos\theta + P_x \sin\theta \end{aligned} \tag{3.13}$$

We can rewrite this in matrix form as follows.

$$\mathbf{P}' = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \mathbf{P} \tag{3.14}$$

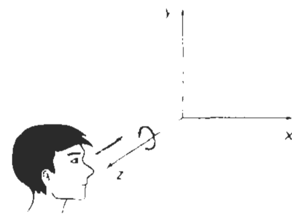
The 2D rotation matrix in Equation (3.14) can be extended to a rotation about the z-axis in three dimensions by taking the third row and column from the identity matrix. This ensures that the z-coordinate of a vector remains fixed during a rotation about the z-axis, as we would expect. The matrix $\mathbf{R}_z(\theta)$ that performs a rotation through the angle θ about the z-axis is thus given by

$$\mathbf{R}_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{3.15}$$

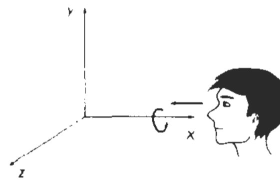
Similarly, we can derive the following 3×3 matrices $\mathbf{R}_x(\theta)$ and $\mathbf{R}_y(\theta)$ that perform rotations through an angle θ about the x- and y-axes, respectively.

$$\mathbf{R}_x(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

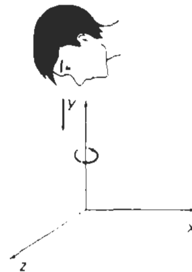
$$\mathbf{R}_y(\theta) = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix} \tag{3.16}$$



(a)



(b)



(c)

Figure 11-3
Positive rotation directions about the coordinate axes are counterclockwise, when looking toward the origin from a positive coordinate position on each axis.

$$
\begin{aligned}
x' &= x\cos\theta - y\sin\theta \\
y' &= x\sin\theta + y\cos\theta \\
z' &= z
\end{aligned}
\tag{11-4}
$$

Parameter θ specifies the rotation angle. In homogeneous coordinate form, the three-dimensional z-axis rotation equations are expressed as

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{11-5}$$

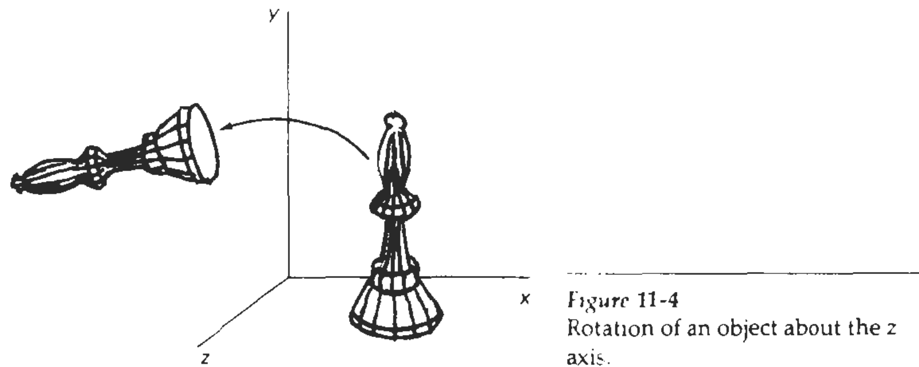


Figure 11-4
Rotation of an object about the z axis.

which we can write more compactly as

$$\mathbf{P}' = \mathbf{R}_z(\theta) \cdot \mathbf{P} \tag{11-6}$$

Figure 11-4 illustrates rotation of an object about the z axis.

Transformation equations for rotations about the other two coordinate axes can be obtained with a cyclic permutation of the coordinate parameters x, y, and z in Eqs. 11-4. That is, we use the replacements

$$x \rightarrow y \rightarrow z \rightarrow x \tag{11-7}$$

as illustrated in Fig. 11-5.

Substituting permutations 11-7 in Eqs. 11-4, we get the equations for an **x-axis rotation:**

$$\begin{aligned} y' &= y\cos\theta - z\sin\theta \\ z' &= y\sin\theta + z\cos\theta \\ x' &= x \end{aligned} \tag{11-8}$$

which can be written in the homogeneous coordinate form

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{11-9}$$

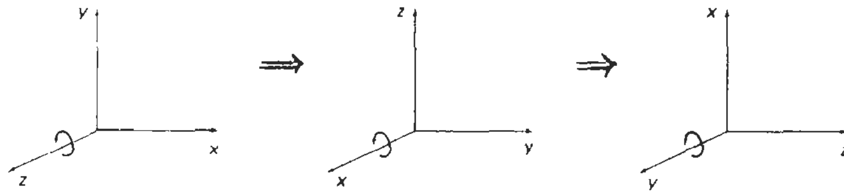


Figure 11-5
Cyclic permutation of the Cartesian-coordinate axes to produce the three sets of coordinate-axis rotation equations.

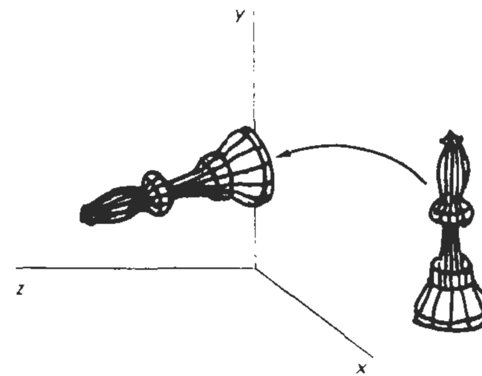


Figure 11-6
Rotation of an object about the x axis.

or

$$\mathbf{P}' = \mathbf{R}_x(\theta) \cdot \mathbf{P} \tag{11-10}$$

Rotation of an object around the x axis is demonstrated in Fig. 11.6.

Cyclically permuting coordinates in Eqs. 11-8 give us the transformation equations for a y-**axis rotation**:

$$\begin{aligned} z' &= z\cos\theta - x\sin\theta \\ x' &= z\sin\theta + x\cos\theta \\ y' &= y \end{aligned} \tag{11-11}$$

The matrix representation for y-axis rotation is

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \tag{11-12}$$

or

$$\mathbf{P}' = \mathbf{R}_y(\theta) \cdot \mathbf{P} \tag{11-13}$$

3.6 Quaternions

A *quaternion* is an alternative mathematical entity that 3D graphics programmers use to represent rotations. The use of quaternions has advantages over the use of rotation matrices in many situations because quaternions require less storage space, concatenation of quaternions requires fewer arithmetic operations, and quaternions are more easily interpolated for producing smooth animation.

3.6.1 Quaternion Mathematics

The set of quaternions, known by mathematicians as the ring of Hamiltonian quaternions and denoted by $\mathbb{H}$, can be thought of as a four-dimensional vector space for which an element $\mathbf{q}$ has the form

$$\mathbf{q} = \langle w, x, y, z \rangle = w + xi + yj + zk. \tag{3.32}$$

A quaternion is often written as $\mathbf{q} = s + \mathbf{v}$, where s represents the scalar part corresponding to the w-component of $\mathbf{q}$, and $\mathbf{v}$ represents the vector part corresponding to the x-, y-, and z-components of $\mathbf{q}$.

The set of quaternions is a natural extension of the set of complex numbers. Multiplication of quaternions is defined using the ordinary distributive law and adhering to the following rules when multiplying the "imaginary" components i, j, and k.

$$\begin{aligned} i^2 = j^2 = k^2 &= -1 \\ ij = -ji &= k \\ jk = -kj &= i \\ ki = -ik &= j \end{aligned} \tag{3.33}$$

Multiplication of quaternions is not commutative, and so we must be careful to multiply terms in the correct order. For two quaternions $\mathbf{q}_1 = w_1 + x_1 i + y_1 j + z_1 k$ and $\mathbf{q}_2 = w_2 + x_2 i + y_2 j + z_2 k$, the product $\mathbf{q}_1\mathbf{q}_2$ is given by

$$\begin{aligned} \mathbf{q}_1\mathbf{q}_2 = &\ (w_1 w_2 - x_1 x_2 - y_1 y_2 - z_1 z_2) \\ &+ (w_1 x_2 + x_1 w_2 + y_1 z_2 - z_1 y_2)i \\ &+ (w_1 y_2 - x_1 z_2 + y_1 w_2 + z_1 x_2)j \\ &+ (w_1 z_2 + x_1 y_2 - y_1 x_2 + z_1 w_2)k. \end{aligned} \tag{3.34}$$

When written in scalar-vector form, the product of two quaternions $\mathbf{q}_1 = s_1 + \mathbf{v}_1$ and $\mathbf{q}_2 = s_2 + \mathbf{v}_2$ can be written as

$$\mathbf{q}_1\mathbf{q}_2 = s_1 s_2 - \mathbf{v}_1 \cdot \mathbf{v}_2 + s_1\mathbf{v}_2 + s_2\mathbf{v}_1 + \mathbf{v}_1 \times \mathbf{v}_2. \tag{3.35}$$

Definition 3.4. The *conjugate* of a quaternion $\mathbf{q} = s + \mathbf{v}$, denoted by $\overline{\mathbf{q}}$, is given by $\overline{\mathbf{q}} = s - \mathbf{v}$.

A short calculation reveals that the product of a quaternion $\mathbf{q}$ and its conjugate $\overline{\mathbf{q}}$ is equal to the dot product of $\mathbf{q}$ with itself, which is also equal to the square of the magnitude of $\mathbf{q}$. That is,

$$\mathbf{q}\overline{\mathbf{q}} = \overline{\mathbf{q}}\mathbf{q} = \mathbf{q} \cdot \mathbf{q} = \|\mathbf{q}\|^2 = q^2. \tag{3.36}$$

Theorem 3.5. The *inverse* of a nonzero quaternion $\mathbf{q}$, denoted by $\mathbf{q}^{-1}$, is given by

$$\mathbf{q}^{-1} = \frac{\overline{\mathbf{q}}}{q^2}. \tag{3.37}$$

Proof. Applying Equation (3.36), we have

$$\mathbf{q}\mathbf{q}^{-1} = \frac{\mathbf{q}\overline{\mathbf{q}}}{q^2} = \frac{q^2}{q^2} = 1 \tag{3.38}$$

and

$$\mathbf{q}^{-1}\mathbf{q} = \frac{\overline{\mathbf{q}}\mathbf{q}}{q^2} = \frac{q^2}{q^2} = 1, \tag{3.39}$$

thus proving the theorem. ■

3.6.2 Rotations with Quaternions

A rotation in three dimensions can be thought of as a function φ that maps $\mathbb{R}^3$ onto itself. For φ to represent a rotation, it must preserve lengths, angles, and handedness. Length preservation is satisfied if

$$\|\varphi(\mathbf{P})\| = \|\mathbf{P}\|. \tag{3.40}$$

The angle between the line segments connecting the origin to any two points $\mathbf{P}_1$ and $\mathbf{P}_2$ is preserved if

$$\varphi(\mathbf{P}_1) \cdot \varphi(\mathbf{P}_2) = \mathbf{P}_1 \cdot \mathbf{P}_2. \tag{3.41}$$

Finally, handedness is preserved if

$$\varphi(\mathbf{P}_1) \times \varphi(\mathbf{P}_2) = \varphi(\mathbf{P}_1 \times \mathbf{P}_2). \tag{3.42}$$

Extending the function φ to a mapping from $\mathbb{H}$ onto itself by requiring that $\varphi(s + \mathbf{v}) = s + \varphi(\mathbf{v})$ allows us to rewrite Equation (3.41) as

$$\varphi(\mathbf{P}_1) \cdot \varphi(\mathbf{P}_2) = \varphi(\mathbf{P}_1 \cdot \mathbf{P}_2). \tag{3.43}$$

Treating $\mathbf{P}_1$ and $\mathbf{P}_2$ as quaternions with zero scalar part enables us to combine Equations (3.42) and (3.43) since $\mathbf{P}_1\mathbf{P}_2 = -\mathbf{P}_1 \cdot \mathbf{P}_2 + \mathbf{P}_1 \times \mathbf{P}_2$. We can therefore write the angle preservation and handedness preservation requirements as the single equation

$$\varphi(\mathbf{P}_1)\varphi(\mathbf{P}_2) = \varphi(\mathbf{P}_1\mathbf{P}_2). \tag{3.44}$$

A function φ that satisfies this equation is called a *homomorphism*.

The class of functions given by

$$\varphi_{\mathbf{q}}(\mathbf{P}) = \mathbf{q}\mathbf{P}\mathbf{q}^{-1}, \tag{3.45}$$

where $\mathbf{q}$ is a nonzero quaternion, satisfies the requirements stated in Equations (3.40) and (3.44), and thus represents a set of rotations. This fact can be proven by first observing that the function $\varphi_{\mathbf{q}}$ preserves lengths because

$$\left\|\varphi_{\mathbf{q}}(\mathbf{P})\right\| = \left\|\mathbf{q}\mathbf{P}\mathbf{q}^{-1}\right\| = \|\mathbf{q}\|\|\mathbf{P}\|\left\|\mathbf{q}^{-1}\right\| = \|\mathbf{P}\|\frac{\|\mathbf{q}\|\|\overline{\mathbf{q}}\|}{q^2} = \|\mathbf{P}\|. \tag{3.46}$$

Furthermore, $\varphi_{\mathbf{q}}$ is a homomorphism since

$$\varphi_{\mathbf{q}}(\mathbf{P}_1)\varphi_{\mathbf{q}}(\mathbf{P}_2) = \mathbf{q}\mathbf{P}_1\mathbf{q}^{-1}\mathbf{q}\mathbf{P}_2\mathbf{q}^{-1} = \mathbf{q}\mathbf{P}_1\mathbf{P}_2\mathbf{q}^{-1} = \varphi_{\mathbf{q}}(\mathbf{P}_1\mathbf{P}_2). \tag{3.47}$$

We now need to find a formula for the quaternion $\mathbf{q}$ corresponding to a rotation through the angle θ about the axis $\mathbf{A}$. A quick calculation shows that $\varphi_{a\mathbf{q}} = \varphi_{\mathbf{q}}$ for any nonzero scalar a, so to keep things as simple as possible, we will concern ourselves only with unit quaternions.

Let $\mathbf{q} = s + \mathbf{v}$ be a unit quaternion. Then $\mathbf{q}^{-1} = s - \mathbf{v}$, and given a point $\mathbf{P}$, we have

$$\begin{aligned}
\mathbf{q}\mathbf{P}\mathbf{q}^{-1} &= (s + \mathbf{v})\mathbf{P}(s - \mathbf{v}) \\
&= (-\mathbf{v} \cdot \mathbf{P} + s\mathbf{P} + \mathbf{v} \times \mathbf{P})(s - \mathbf{v}) \\
&= -s\mathbf{v} \cdot \mathbf{P} + s^2\mathbf{P} + s\mathbf{v} \times \mathbf{P} + (\mathbf{v} \cdot \mathbf{P})\mathbf{v} - s\mathbf{P}\mathbf{v} - (\mathbf{v} \times \mathbf{P})\mathbf{v} \\
&= s^2\mathbf{P} + 2s\mathbf{v} \times \mathbf{P} + (\mathbf{v} \cdot \mathbf{P})\mathbf{v} - \mathbf{v} \times \mathbf{P} \times \mathbf{v}.
\end{aligned} \tag{3.48}$$

When we compare this to the formula for rotation about an arbitrary axis given in Equation (3.20), we can infer the following equalities.

$$\begin{aligned} s^2 - t^2 &= \cos\theta \\ 2st &= \sin\theta \\ 2t^2 &= 1 - \cos\theta \end{aligned} \tag{3.51}$$

The third equality gives us

$$t = \sqrt{\frac{1 - \cos\theta}{2}} = \sin\frac{\theta}{2}. \tag{3.52}$$

The first and third equalities together tell us that $s^2 + t^2 = 1$, so we must have $s = \cos(\theta/2)$. (The fact that $\sin 2\theta = 2\sin\theta\cos\theta$ verifies that the second equality is satisfied by these values for s and t.)

We have now determined that the unit quaternion **q** corresponding to a rotation through the angle θ about the axis **A** is given by

$$\mathbf{q} = \cos\frac{\theta}{2} + \mathbf{A}\sin\frac{\theta}{2}. \tag{3.53}$$

It is often necessary to convert a quaternion into the equivalent 3×3 rotation matrix, for instance, to pass the transform for an object to a 3D graphics library. We can determine the formula for the matrix corresponding to the quaternion $\mathbf{q} = s + t\mathbf{A}$ by using Equations (1.25) and (1.20) to write Equation (3.50) in matrix form. (This is nearly identical to the technique used in Section 3.3.1.) This gives us

$$\mathbf{qPq}^{-1} = \begin{bmatrix} s^2 - t^2 & 0 & 0 \\ 0 & s^2 - t^2 & 0 \\ 0 & 0 & s^2 - t^2 \end{bmatrix} \mathbf{P} + \begin{bmatrix} 0 & -2stA_z & 2stA_y \\ 2stA_z & 0 & -2stA_x \\ -2stA_y & 2stA_x & 0 \end{bmatrix} \mathbf{P}$$
$$+ \begin{bmatrix} 2t^2A_x^2 & 2t^2A_xA_y & 2t^2A_xA_z \\ 2t^2A_xA_y & 2t^2A_y^2 & 2t^2A_yA_z \\ 2t^2A_xA_z & 2t^2A_yA_z & 2t^2A_z^2 \end{bmatrix} \mathbf{P}. \tag{3.56}$$

Writing the quaternion $\mathbf{q}$ as the four-dimensional vector $\mathbf{q} = \langle w, x, y, z \rangle$, we have $w = s$, $x = tA_x$, $y = tA_y$, and $z = tA_z$. Since $\mathbf{A}$ is a unit vector,

$$x^2 + y^2 + z^2 = t^2A^2 = t^2. \tag{3.57}$$

Rewriting Equation (3.56) in terms of the components w, x, y, and z gives us

$$\mathbf{qPq}^{-1} = \begin{bmatrix} w^2 - x^2 - y^2 - z^2 & 0 & 0 \\ 0 & w^2 - x^2 - y^2 - z^2 & 0 \\ 0 & 0 & w^2 - x^2 - y^2 - z^2 \end{bmatrix} \mathbf{P}$$
$$+ \begin{bmatrix} 0 & -2wz & 2wy \\ 2wz & 0 & -2wx \\ -2wy & 2wx & 0 \end{bmatrix} \mathbf{P} + \begin{bmatrix} 2x^2 & 2xy & 2xz \\ 2xy & 2y^2 & 2yz \\ 2xz & 2yz & 2z^2 \end{bmatrix} \mathbf{P}. \tag{3.58}$$

Since **q** is a unit quaternion, we know that $w^2 + x^2 + y^2 + z^2 = 1$, so we can write

$$w^2 - x^2 - y^2 - z^2 = 1 - 2x^2 - 2y^2 - 2z^2. \tag{3.59}$$

Using this equation and combining the three matrices gives us the following formula for the matrix $\mathbf{R}_q$, the rotation matrix corresponding to the quaternion **q**.

$$\mathbf{R}_q = \begin{bmatrix} 1 - 2y^2 - 2z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & 1 - 2x^2 - 2z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & 1 - 2x^2 - 2y^2 \end{bmatrix}$$

Why use quaternions

Quaternions have some advantages over other representations of rotations.

- Quaternions don't suffer from gimbal lock, unlike Euler angles.
- They can be represented as 4 numbers, in contrast to the 9 numbers of a rotations matrix.
- The conversion to and from axis/angle representation is trivial.
- Smooth interpolation between two quaternions is easy (in contrast to axis/angle or rotation matrices).
- After a lot of calculations on quaternions and matrices, rounding errors accumulate, so you have to normalize quaternions and orthogonalize a rotation matrix, but normalizing a quaternion is a lot less troublesome than orthogonalizing a matrix.
- Similar to rotation matrices, you can just multiply 2 quaternions together to receive a quaternion that represents both rotations.

The only disadvantages of quaternions are:

- They are hard to visualize.
- You have to convert them to get a human-readable representation (Euler angles) or something OpenGL can understand (Matrix).
- Smooth interpolation between quaternions is complicated by the fact that each 3D rotation has two representations.

Multiplying quaternions

To multiply two quaternions, write each one as the sum of a scalar and a vector. The product of $q_1 = w_1 + \vec{v_1}$ and $q_2 = w_2 + \vec{v_2}$ is $q = w + \vec{v}$ where

$$w = w_1 w_2 - \vec{v_1} \cdot \vec{v_2}$$
$$\vec{v} = w_1 \vec{v_2} + w_2 \vec{v_1} + \vec{v_1} \times \vec{v_2}$$

```
// Multiplying q1 with q2 applies the rotation q2 to q1
Quaternion Quaternion::operator* (const Quaternion &rq) const
{
	// the constructor takes its arguments as (x, y, z, w)
	return Quaternion(w * rq.x + x * rq.w + y * rq.z - z * rq.y,
	                  w * rq.y + y * rq.w + z * rq.x - x * rq.z,
	                  w * rq.z + z * rq.w + x * rq.y - y * rq.x,
	                  w * rq.w - x * rq.x - y * rq.y - z * rq.z);
}
```

Please note: Quaternion-multiplication is NOT commutative. Thus q1 * q2 is not the same as q2 * q1. This is pretty obvious actually: As I explained, quaternions represent rotations and multiplying them "concatenates" the rotations. Now take you hand and hold it parallel to the floor so your hand points away from you. Rotate it 90° around the x-axis so it is pointing upward. Now rotate it 90° clockwise around its local y-axis (the one coming out of the back of your hand). Your hand should now be pointing to your right, with you looking at the back of your hand. Now invert the rotations: Rotate your hand around the y-axis so its facing right with the back of the hand facing upwards. Now rotate around the x axis and your hand is pointing up, back of hand facing your left. See, the order in which you apply rotations matters. Ok, ok, you probably knew that...

Rotating vectors

To apply a quaternion-rotation to a vector, you need to multiply the vector by the quaternion and its conjugate.

$$\vec{v}' = q\,\vec{v}\,\overline{q}$$

```
// Multiplying a quaternion q with a vector v applies the q-rotation to v
Vector3 Quaternion::operator* (const Vector3 &vec) const
{
	Vector3 vn(vec);
	vn.normalise();

	Quaternion vecQuat, resQuat;
	vecQuat.x = vn.x;
	vecQuat.y = vn.y;
	vecQuat.z = vn.z;
	vecQuat.w = 0.0f;

	resQuat = vecQuat * getConjugate();
	resQuat = *this * resQuat;

	return (Vector3(resQuat.x, resQuat.y, resQuat.z));
}
```

Quaternion from Euler angles

```
// Convert from Euler Angles
void Quaternion::FromEuler(float pitch, float yaw, float roll)
{
        // Basically we create 3 Quaternions, one for pitch, one for yaw, one for roll
        // and multiply those together.
        // the calculation below does the same, just shorter

        float p = pitch * PIOVER180 / 2.0;
        float y = yaw * PIOVER180 / 2.0;
        float r = roll * PIOVER180 / 2.0;

        float sinp = sin(p);
        float siny = sin(y);
        float sinr = sin(r);
        float cosp = cos(p);
        float cosy = cos(y);
        float cosr = cos(r);

        this->x = sinr * cosp * cosy - cosr * sinp * siny;
        this->y = cosr * sinp * cosy + sinr * cosp * siny;
        this->z = cosr * cosp * siny - sinr * sinp * cosy;
        this->w = cosr * cosp * cosy + sinr * sinp * siny;

        normalise();
}
```

Quaternion to Matrix

```
// Convert to Matrix
Matrix4 Quaternion::getMatrix() const
{
        float x2 = x * x;
        float y2 = y * y;
        float z2 = z * z;
        float xy = x * y;
        float xz = x * z;
        float yz = y * z;
        float wx = w * x;
        float wy = w * y;
        float wz = w * z;

        // This calculation would be a lot more complicated for non-unit length quaternions
        // Note: The constructor of Matrix4 expects the Matrix in column-major format like expecte
        //   OpenGL
        return Matrix4( 1.0f - 2.0f * (y2 + z2), 2.0f * (xy - wz), 2.0f * (xz + wy), 0.0f,
                        2.0f * (xy + wz), 1.0f - 2.0f * (x2 + z2), 2.0f * (yz - wx), 0.0f,
                        2.0f * (xz - wy), 2.0f * (yz + wx), 1.0f - 2.0f * (x2 + y2), 0.0f,
                        0.0f, 0.0f, 0.0f, 1.0f)
}
```

```
void Camera::movex(float xmmod)
{
        pos += rotation * Vector3(xmmod, 0.0f, 0.0f);
}

void Camera::movey(float ymmod)
{
        pos.y -= ymmod;
}

void Camera::movez(float zmmod)
{
        pos += rotation * Vector3(0.0f, 0.0f, -zmmod);
}

void Camera::rotatex(float xrmod)
{
        Quaternion nrot(Vector3(1.0f, 0.0f, 0.0f), xrmod * PIOVER180);
        rotation = rotation * nrot;
}

void Camera::rotatey(float yrmod)
{
        Quaternion nrot(Vector3(0.0f, 1.0f, 0.0f), yrmod * PIOVER180);
        rotation = nrot * rotation;
}

void Camera::tick(float seconds)
{
        if (xrot != 0.0f) rotatex(xrot * seconds * rotspeed);
        if (yrot != 0.0f) rotatey(yrot * seconds * rotspeed);

        if (xmov != 0.0f) movex(xmov * seconds * movespeed);
        if (ymov != 0.0f) movey(ymov * seconds * movespeed);
        if (zmov != 0.0f) movez(zmov * seconds * movespeed);
}
```

OpenGL Geometric Drawing Primitives

Now that you've seen how to specify vertices, you still need to know how to tell OpenGL to create a set of points, a line, or a polygon from those vertices. To do this, you bracket each set of vertices between a call to **glBegin()** and a call to **glEnd()**. The argument passed to **glBegin()** determines what sort of geometric primitive is constructed from the vertices. For example, Example 2-3 specifies the vertices for the polygon shown in Figure 2-6.

Example 2-3 : Filled Polygon

```
glBegin(GL_POLYGON);
   glVertex2f(0.0, 0.0);
   glVertex2f(0.0, 3.0);
   glVertex2f(4.0, 3.0);
   glVertex2f(6.0, 1.5);
   glVertex2f(4.0, 0.0);
glEnd();
```

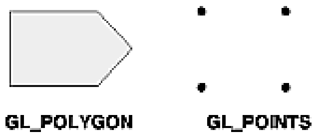


Figure 2-6 : Drawing a Polygon or a Set of Points

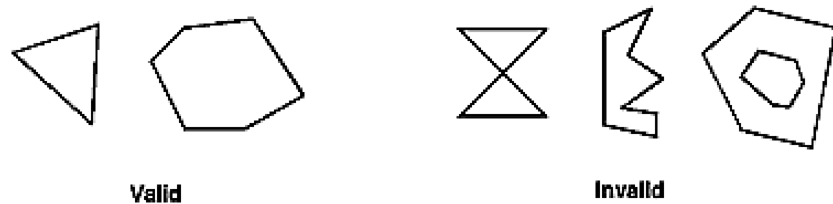


Figure 2-3 : Valid and Invalid Polygons

The reason for the OpenGL restrictions on valid polygon types is that it’s simpler to provide fast polygon-rendering hardware for that restricted class of polygons. Simple polygons can be rendered quickly. The difficult cases are hard to detect quickly. So for maximum performance, OpenGL crosses its fingers and assumes the polygons are simple.

Many real-world surfaces consist of nonsimple polygons, nonconvex polygons, or polygons with holes. Since all such polygons can be formed from unions of simple convex polygons, some routines to build more complex objects are provided in the GLU library. These routines take complex descriptions and tessellate them, or break them down into groups of the simpler OpenGL polygons that can then be rendered. (See "Polygon Tessellation" in Chapter 11 for more information about the tessellation routines.)

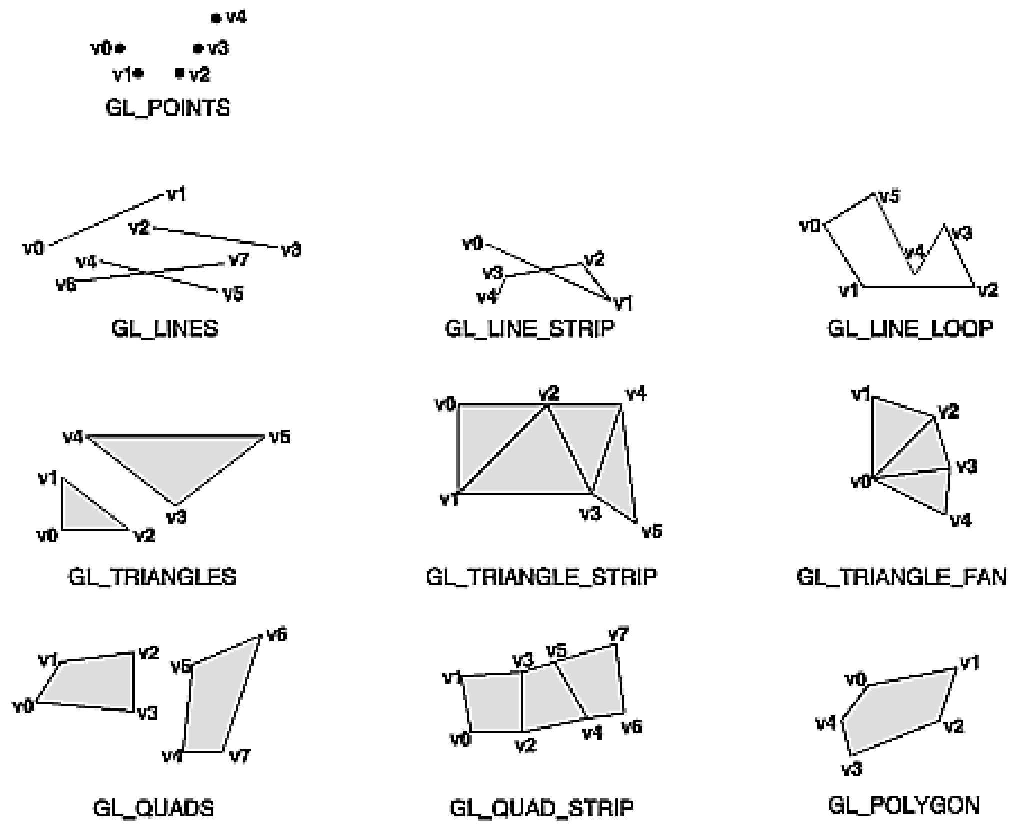


Figure 2-7 : Geometric Primitive Types

As you read the following descriptions, assume that *n* vertices (v0, v1, v2, ... , vn-1) are described between a **glBegin()** and **glEnd()** pair.

GL_POINTS	Draws a point at each of the *n* vertices.
GL_LINES	Draws a series of unconnected line segments. Segments are drawn between v0 and v1, between v2 and v3, and so on. If *n* is odd, the last segment is drawn between vn-3 and vn-2, and vn-1 is ignored.
GL_LINE_STRIP	Draws a line segment from v0 to v1, then from v1 to v2, and so on, finally drawing the segment from vn-2 to vn-1. Thus, a total of *n-1* line segments are drawn. Nothing is drawn unless *n* is larger than 1. There are no restrictions on the vertices describing a line strip (or a line loop); the lines can intersect arbitrarily.
GL_LINE_LOOP	Same as GL_LINE_STRIP, except that a final line segment is drawn from vn-1 to v0, completing a loop.
GL_TRIANGLES	Draws a series of triangles (three-sided polygons) using vertices v0, v1, v2, then v3, v4, v5, and so on. If *n* isn't an exact multiple of 3, the final one or two vertices are ignored.
GL_TRIANGLE_STRIP	Draws a series of triangles (three-sided polygons) using vertices v0, v1,

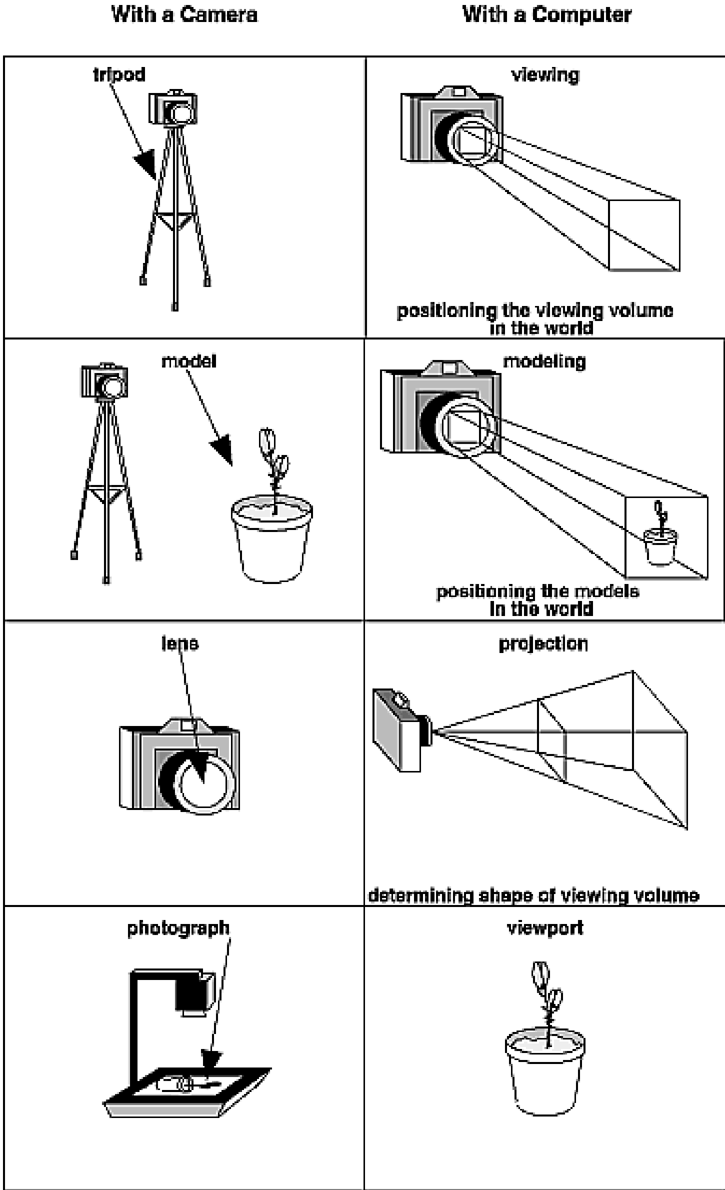


Figure 3-1 : The Camera Analogy

A Simple Example: Drawing a Cube

Example 3-1 draws a cube that's scaled by a modeling transformation (see Figure 3-3). The viewing transformation, **gluLookAt()**, positions and aims the camera towards where the cube is drawn. A projection transformation and a viewport transformation are also specified. The rest of this section walks you through Example 3-1 and briefly explains the transformation commands it uses. The succeeding sections contain the complete, detailed discussion of all OpenGL's transformation commands.

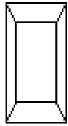


Figure 3-3 : Transformed Cube

Example 3-1 : Transformed Cube: cube.c

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

void init(void)
{
   glClearColor (0.0, 0.0, 0.0, 0.0);
   glShadeModel (GL_FLAT);
}

void display(void)
{
   glClear (GL_COLOR_BUFFER_BIT);
   glColor3f (1.0, 1.0, 1.0);
   glLoadIdentity ();             /* clear the matrix */
           /* viewing transformation  */
   gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
   glScalef (1.0, 2.0, 1.0);      /* modeling transformation */
   glutWireCube (1.0);
   glFlush ();
}

void reshape (int w, int h)
{
   glViewport (0, 0, (GLsizei) w, (GLsizei) h);
   glMatrixMode (GL_PROJECTION);
   glLoadIdentity ();
   glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
   glMatrixMode (GL_MODELVIEW);
}

int main(int argc, char** argv)
{
   glutInit(&argc, argv);
   glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
   glutInitWindowSize (500, 500);
   glutInitWindowPosition (100, 100);
   glutCreateWindow (argv[0]);
   init ();
   glutDisplayFunc(display);
   glutReshapeFunc(reshape);
   glutMainLoop();
   return 0;
}
```

clipping planes, thereby truncating the pyramid. Note that **gluPerspective()** is limited to creating frustums that are symmetric in both the *x*- and *y*-axes along the line of sight, but this is usually what you want.

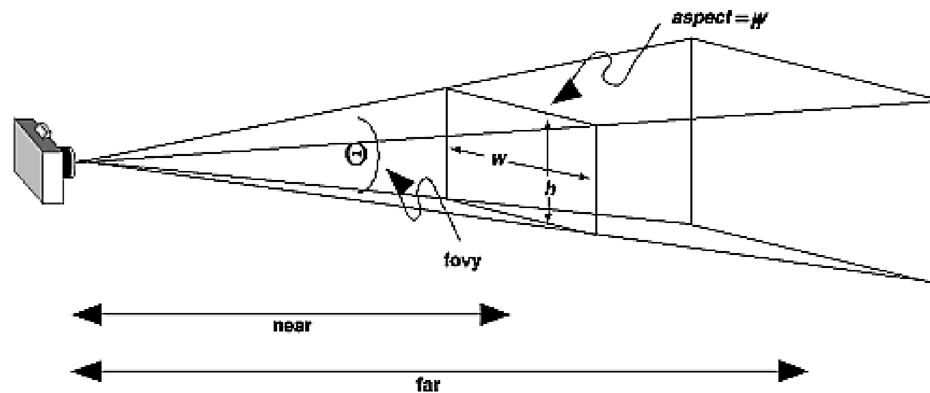


Figure 3-14 : Perspective Viewing Volume Specified by gluPerspective()

void ***gluPerspective****(GLdouble fovy, GLdouble aspect,*
GLdouble near, GLdouble far);

> *Creates a matrix for a symmetric perspective-view frustum and multiplies the current matrix by it. fovy is the angle of the field of view in the x-z plane; its value must be in the range [0.0,180.0]. aspect is the aspect ratio of the frustum, its width divided by its height. near and far values the distances between the viewpoint and the clipping planes, along the negative z-axis. They should always be positive.*

With an orthographic projection, the viewing volume is a rectangular parallelepiped, or more informally, a box (see Figure 3-15). Unlike perspective projection, the size of the viewing volume doesn't change from one end to the other, so distance from the camera doesn't affect how large an object appears. This type of projection is used for applications such as creating architectural blueprints and computer-aided design, where it's crucial to maintain the actual sizes of objects and angles between them as they're projected.

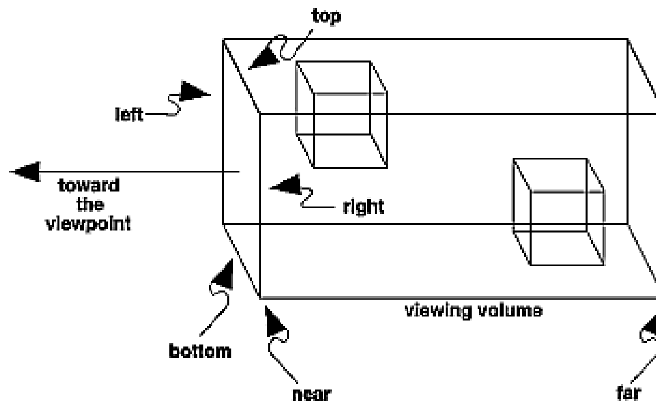


Figure 3-15 : Orthographic Viewing Volume

The command **glOrtho()** creates an orthographic parallel viewing volume. As with **glFrustum()**, you specify the corners of the near clipping plane and the distance to the far clipping plane.

void ***glOrtho****(GLdouble left, GLdouble right, GLdouble bottom,*
GLdouble top, GLdouble near, GLdouble far);

> *Creates a matrix for an orthographic parallel viewing volume and multiplies the current matrix by it. (left, bottom, -near) and (right, top, -near) are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewport window, respectively. (left, bottom, -far) and (right, top, -far) are points on the far clipping plane that are mapped to the same respective corners of the viewport. Both near and far can be positive or negative.*

With no other transformations, the direction of projection is parallel to the *z*-axis, and the viewpoint faces toward the negative *z*-axis. Note that this means that the values passed in for *far* and *near* are used as negative *z* values if these planes are in front of the viewpoint, and positive if they're behind the viewpoint.

Some links:

- http://www.glprogramming.com/manpages/opengl-quick-reference-card.pdf
- The Official Guide to Learning OpenGL, Version 1.1 www.glprogramming.com/red/index.html
- OpenGL Tutorial
- http://www.loria.fr/~roegel/cours/iut/opengl/addison.pdf