Graphics Programming using OpenGL

Why OpenGL?

- Device independence
- Platform independence
  - SGI Irix, Linux, Windows
- Abstractions (GL, GLU, GLUT)
- Open source
- Hardware-independent software interface
- Support of client-server protocol
- Other APIs
  - OpenInventor (object-oriented toolkit)
  - DirectX (Microsoft), Java3D (Sun)

Brief Overview of OpenGL

OpenGL is a software interface that allows the programmer to create 2D and 3D graphics images. OpenGL is both a standard API and the implementation of that API. You can call the functions that comprise OpenGL from a program you write and expect to see the same results no matter where your program is running.

OpenGL is independent of the hardware, operating, and windowing systems in use. The fact that it is windowing-system independent, makes it portable. OpenGL program must interface with the windowing system of the platform where the graphics are to be displayed. Therefore, a number of windowing toolkits have been developed for use with OpenGL.

OpenGL functions in a client/server environment. That is, the application program producing the graphics may run on a machine other than the one on which the graphics are displayed.The server part of OpenGL, which runs on the workstation where the graphics are displayed, can access whatever physical graphics device or frame buffer is available on that machine.

OpenGL's rendering commands, however are "primitive". You can tell the program to draw points, lines, and polygons, and you have to build more complex entities upon these. There are no special-purpose functions that you can call to create graphs, contour plots, maps, or any of the other elements we are used to getting from "old standby programs”. With OpenGL, you have to build these things up yourself.

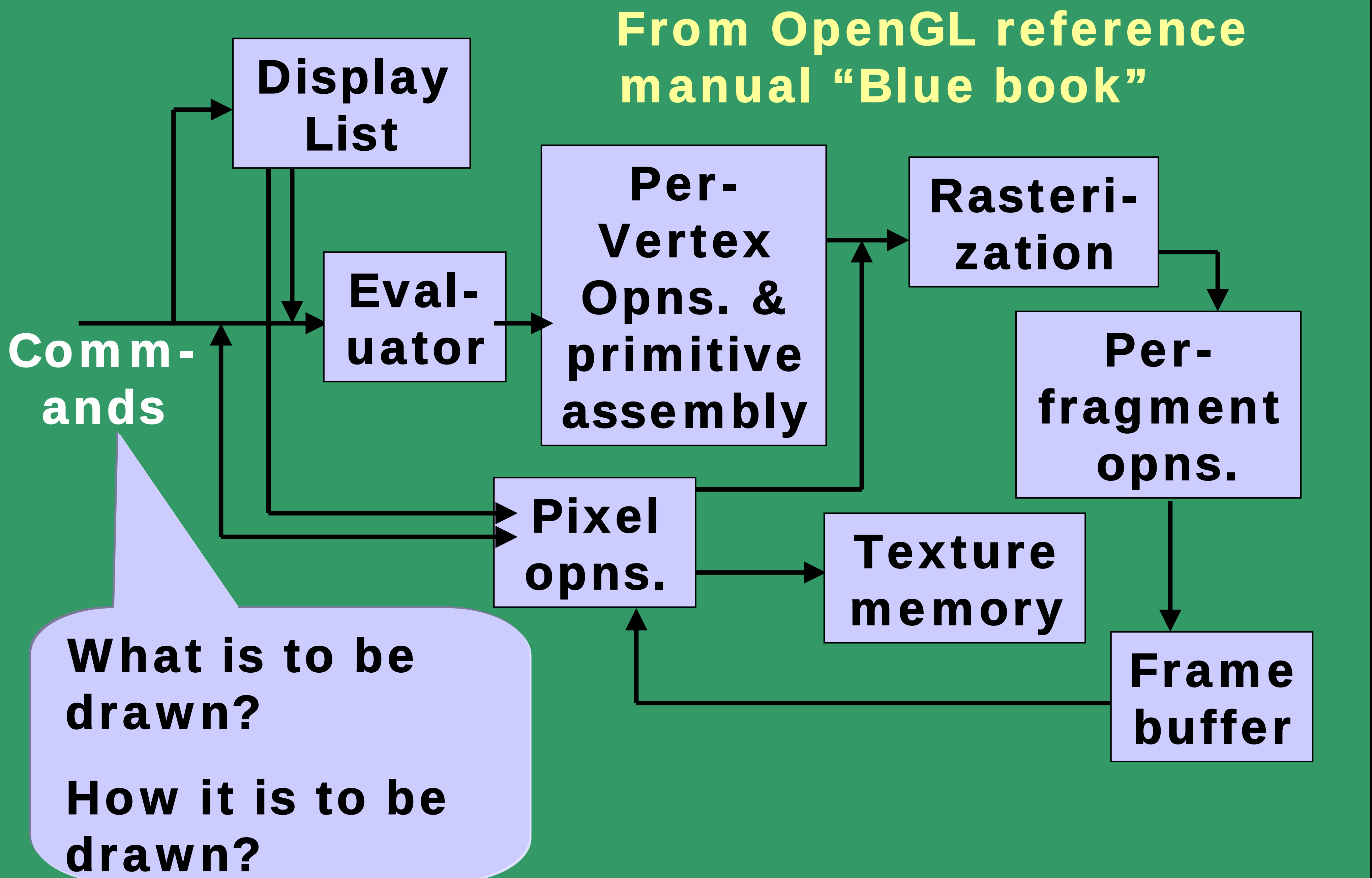
With OpenGL any commands that you execute are executed immediately. That is, when you tell the program to draw something, it does it right away. You also have the option of putting commands into display lists. A display list is a not-editable list of OpenGL commands stored for later execution. You can execute the same display list more than once. For example, you can use display lists to redraw the graphics whenever the user resizes the window. You can use a display list to draw the same shape more than once if it repeats as an element of the picture.

OpenGL is hardware-independent. Many different vendors have written implementations that run on different hardware. These implementations are all written to the same OpenGL standard and are required to pass strict conformance tests. Vendors with licenses include *SGI, AT&T, DEC, Evans & Sutherland, Hitachi, IBM, Intel, Intergraph, Kendall Square Research, Kubota Pacific, Microsoft, NEC, and RasterOps*. The RS/6000 version comes with X and Motif extensions. However X is not required to run OpenGL since OpenGL also runs with other windowing systems.

Features in OpenGL

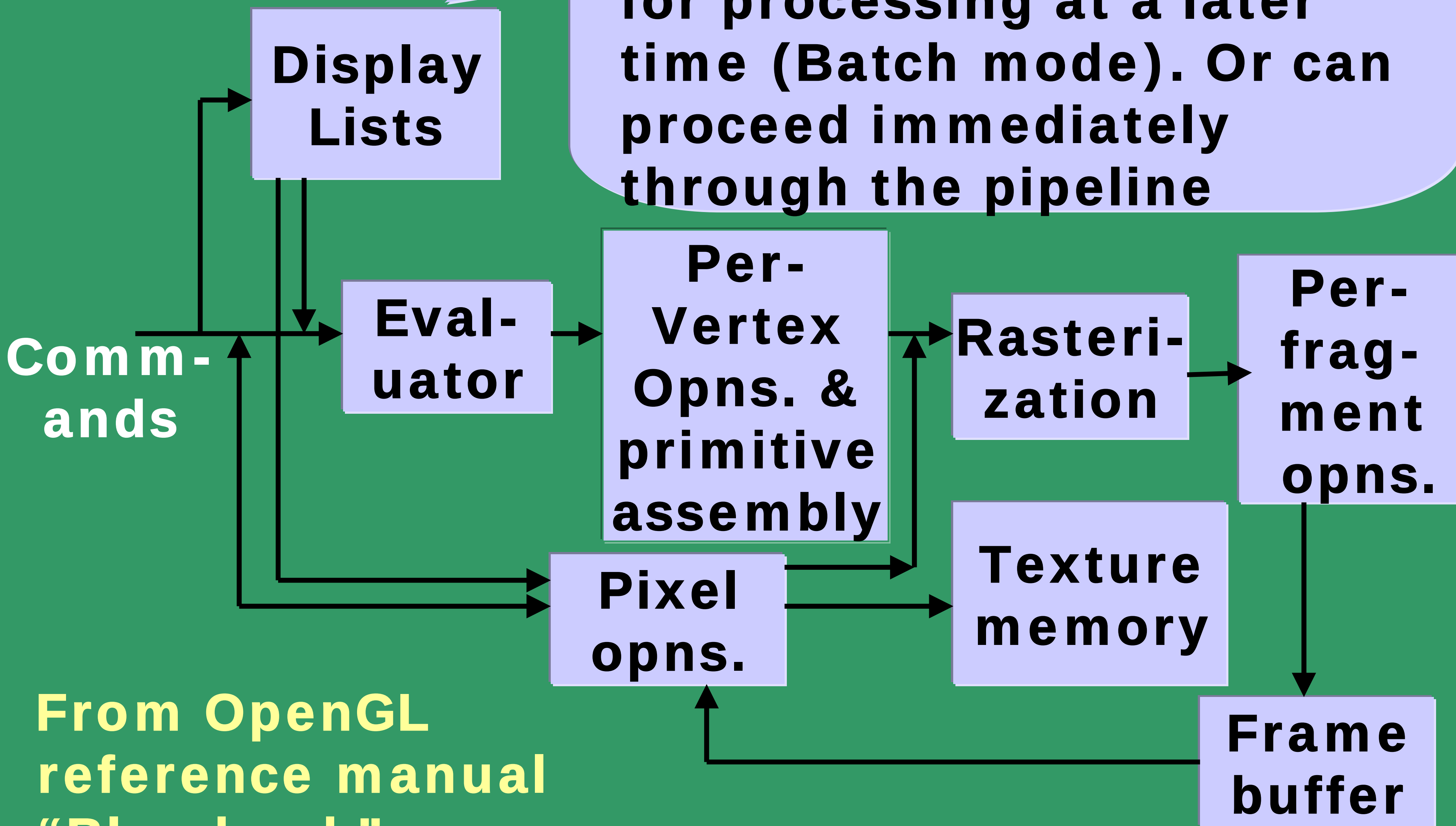
- **3D Transformations**
  - Rotations, scaling, translation, perspective
- **Colour models**
  - Values: R, G, B, alpha.
- **Lighting**
  - Flat shading, Gouraud shading, Phong shading
- **Rendering**
  - Texture mapping
- **Modeling**
  - non-uniform rational B-spline (NURB) curves, surfaces
- **Others**
  - atmospheric fog, alpha blending, motion blur

OpenGL Operation



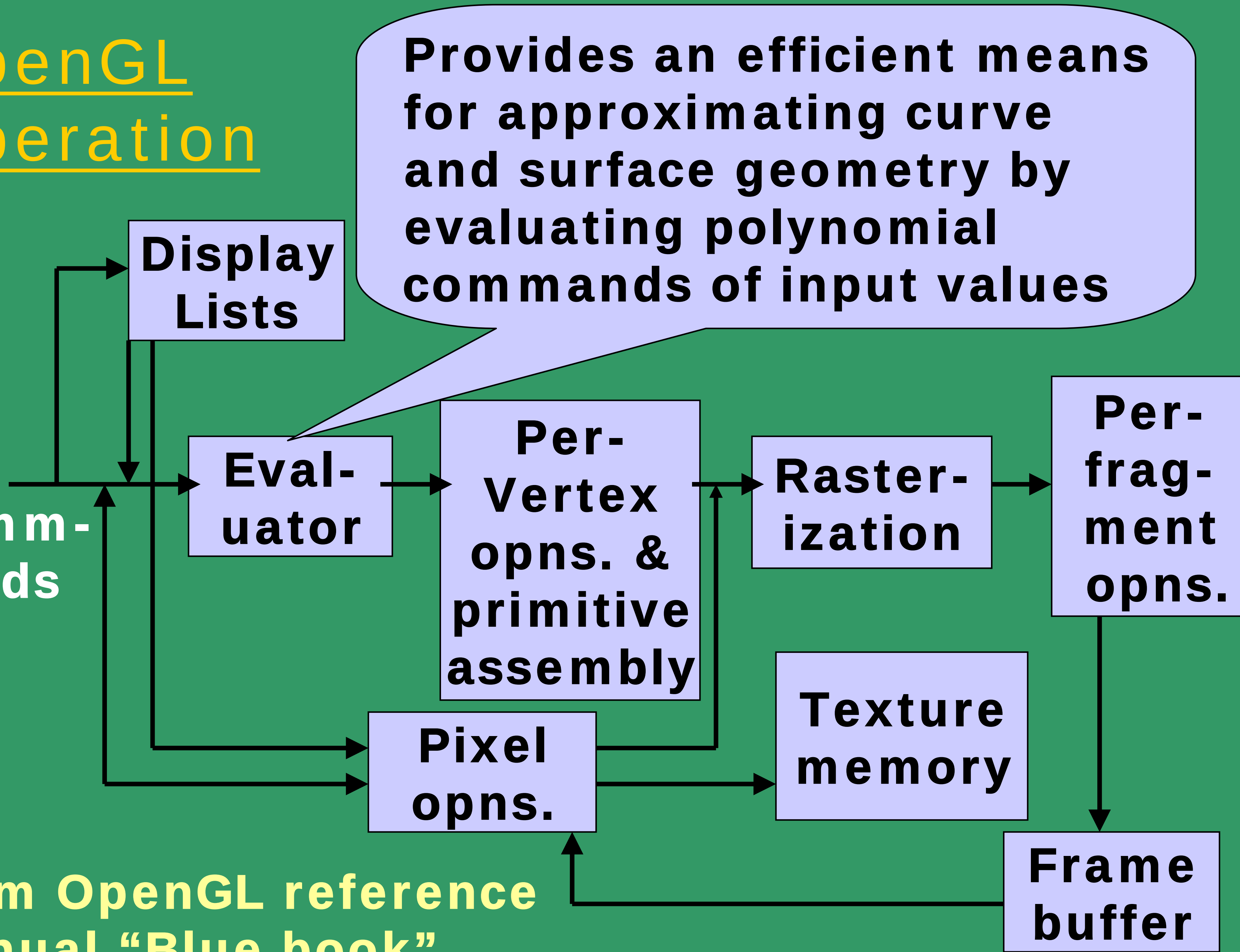
OpenGL Operation

Can accumulate some commands in a display list for processing at a later time (Batch mode). Or can proceed immediately through the pipeline



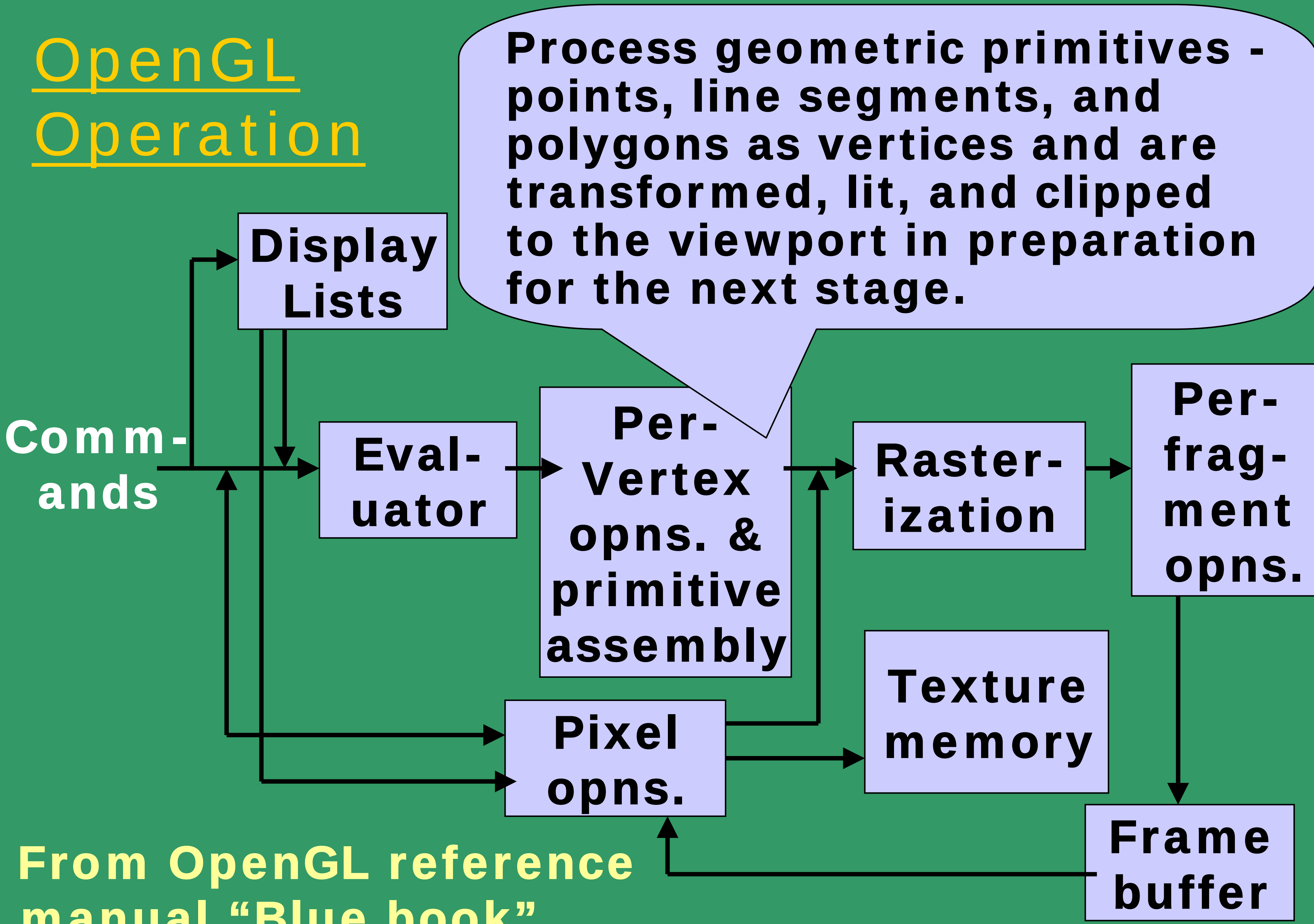
From OpenGL reference manual “Blue book”

OpenGL Operation



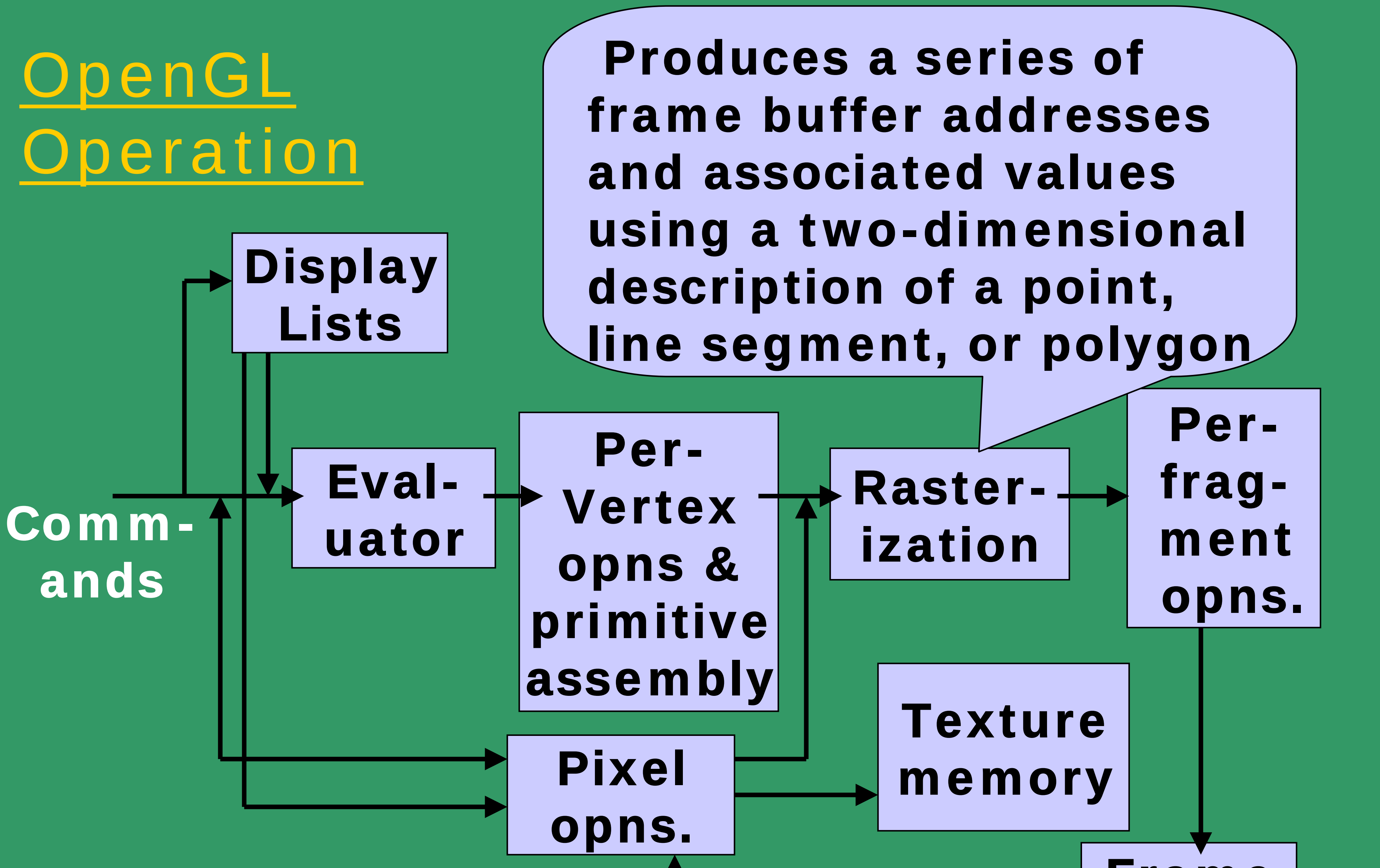
From OpenGL reference manual "Blue book"

OpenGL Operation



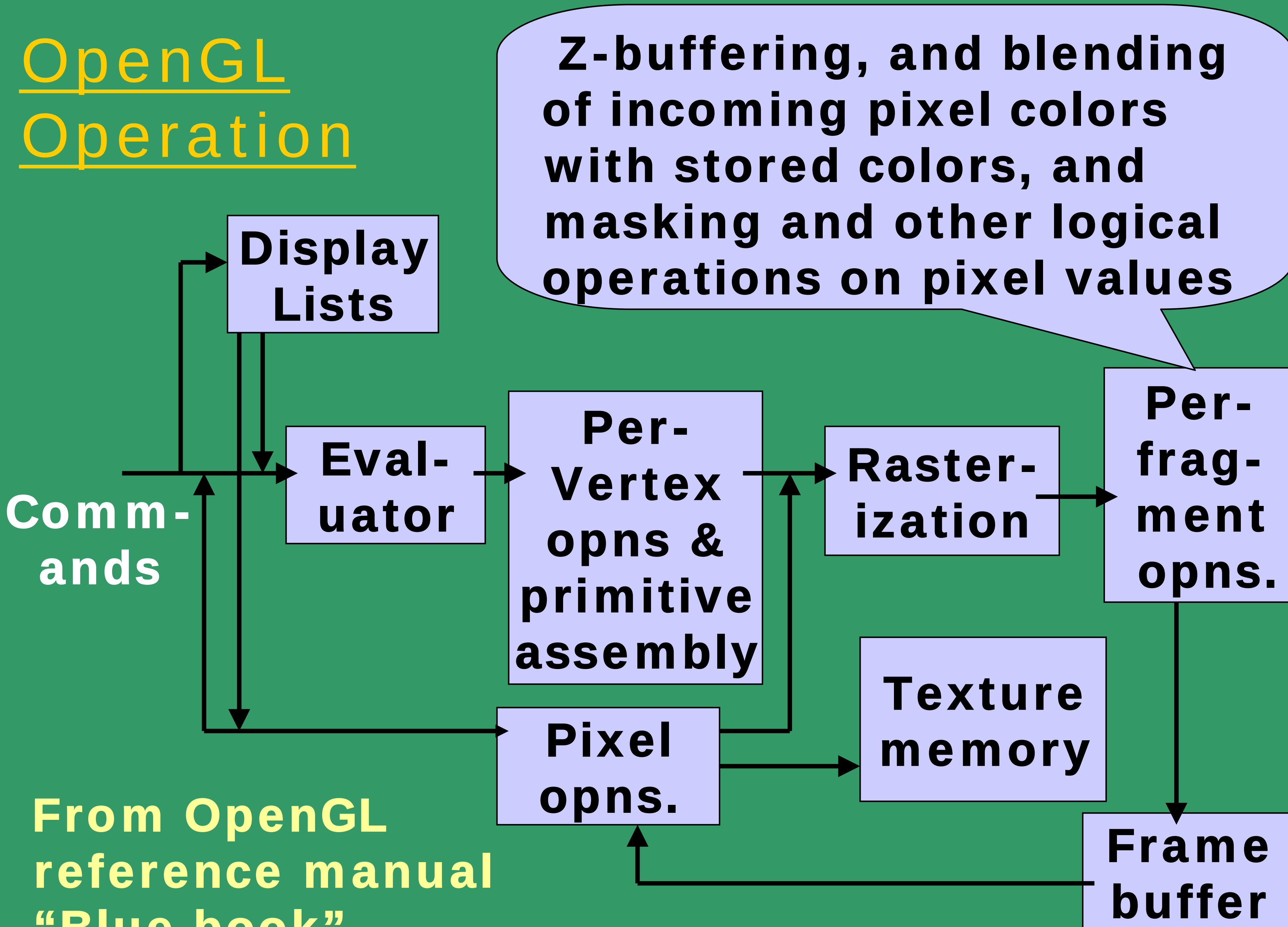
From OpenGL reference manual “Blue book”

OpenGL Operation



From OpenGL reference manual “Blue book”

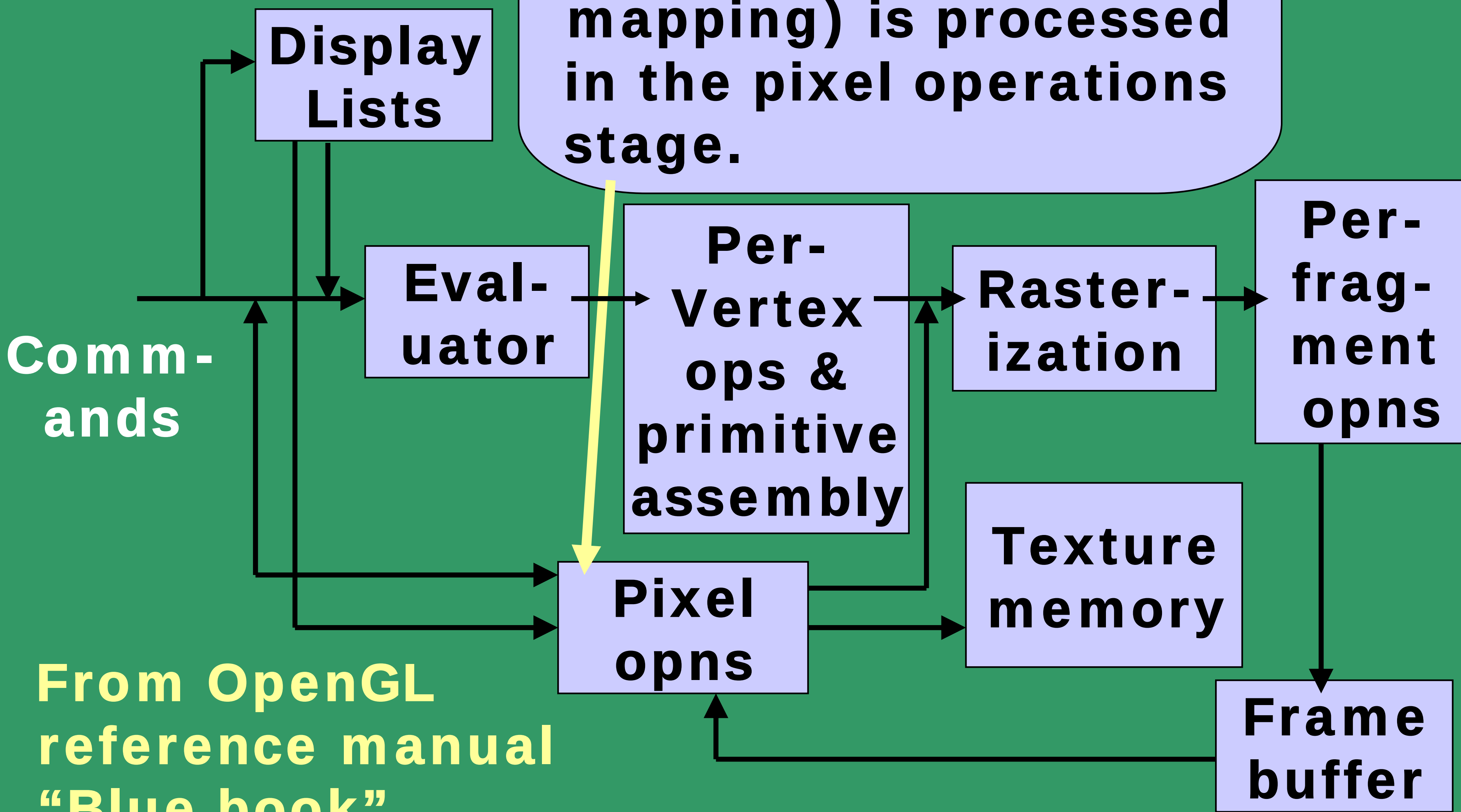
OpenGL Operation



From OpenGL reference manual "Blue book"

OpenGL Operation

Input data can be in the form of pixels (image for texture mapping) is processed in the pixel operations stage.



From OpenGL reference manual "Blue book"

OpenGL Operation

Geometric data (vertices, lines, and polygons) follows the path through the row of boxes that includes evaluators and per-vertex operations, while pixel data (pixels, images, and bitmaps) is treated differently for part of the process.

Both types of data undergo the rasterization and per-fragment operations before the final pixel data is written into the frame buffer.

In the per-vertex operations stage of processing, each vertex's spatial coordinates are transformed by the *modelview* matrix, while the normal vector is transformed by that matrix's inverse and renormalized if specified.

The rasterization process produces fragments (not pixels directly), which consists of color, depth and a texture.

Tests and processing are performed on fragments before they are written into the frame buffer as pixel values.

Abstractions

GLUT

- Windowing toolkit (key, mouse handler, window events)

GLU

- Viewing –perspective / orthographic
- Image scaling, polygon tessellation
- Sphere, cylinders, quadratic surfaces

GL

- Primitives - points, line, polygons
- Shading and Colour
- Translation, rotation, scaling
- Viewing, Clipping, Texture
- Hidden surface removal

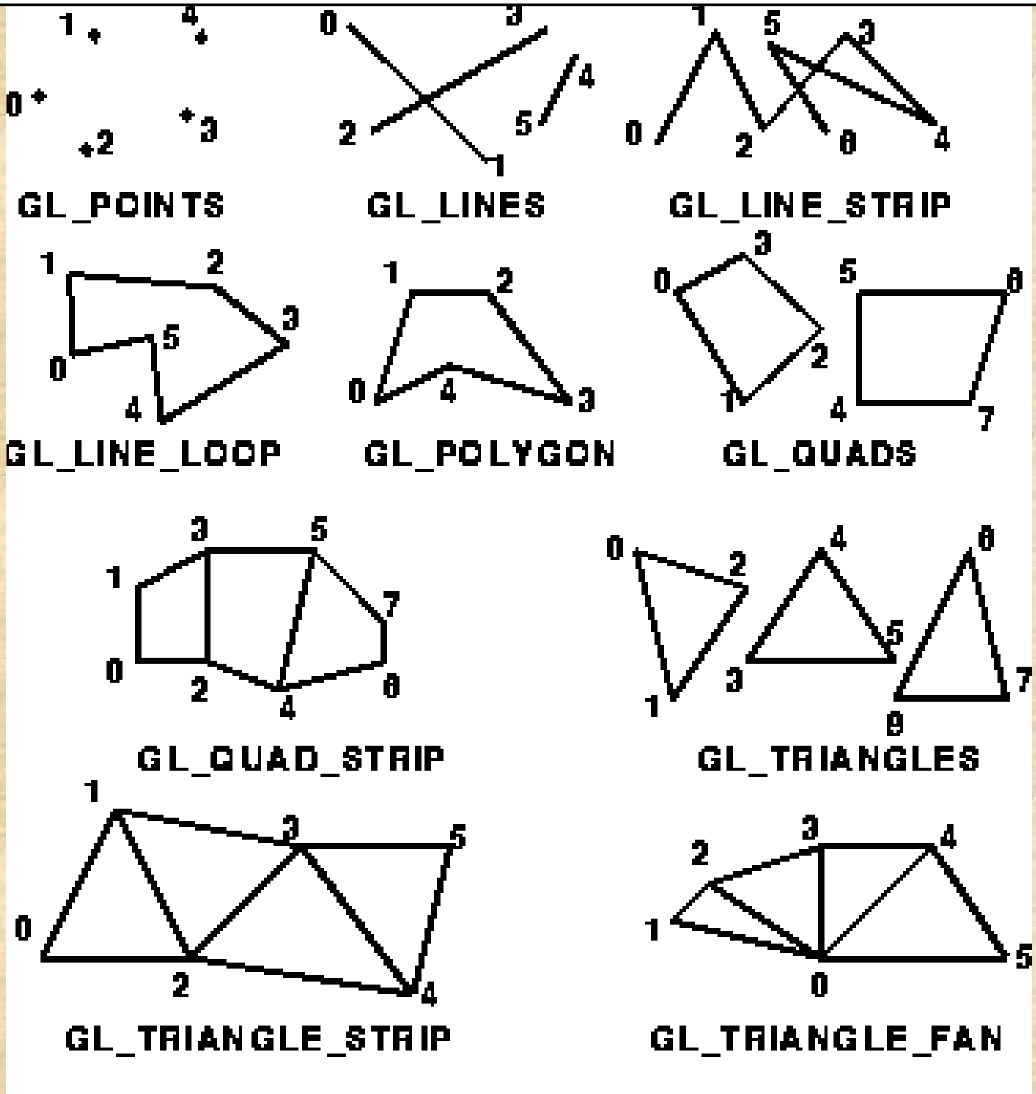
OpenGL Drawing Primitives

OpenGL supports several basic primitive types, including points, lines, quadrilaterals, and general polygons. All of these primitives are specified using a sequence of vertices.

```
glVertex2i(Glint xi, Glint yi);
glVertex3f(Glfloat x, Glfloat y, Glfloat z);
Glfloat vertex[3];

glBegin(GL_LINES);
        glVertex2f(x1, y1);
        glVertex2f(x2, y2);
glEND();
```

Define a pair of points as:

```
glBegin(GL_POINTS);
        glVertex2f(x1, y1);
        glVertex2f(x2, y2);
glEND();
```



The numbers indicate the order in which the vertices have been specified. Note that for the GL_LINES primitive only every second vertex causes a line segment to be drawn. Similarly, for the GL_TRIANGLES primitive, every third vertex causes a triangle to be drawn. Note that for the GL_TRIANGLE_STRIP and GL_TRIANGLE_FAN primitives, a new triangle is produced for every additional vertex. All of the closed primitives shown below are solid-filled, with the exception of GL_LINE_LOOP, which only draws lines connecting the vertices.

The following code fragment illustrates an example of how the primitive type is specified and how the sequence of vertices are passed to OpenGL. It assumes that a window has already been opened and that an appropriate 2D coordinate system has already been established.

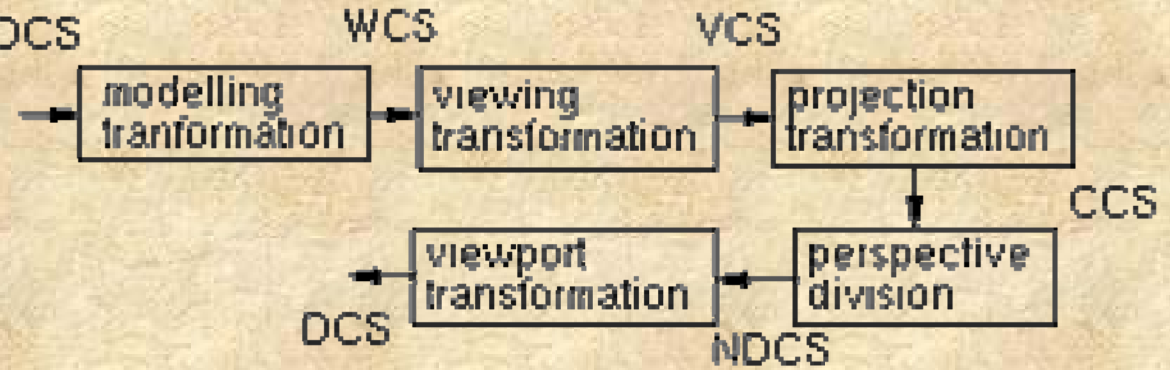
```
// draw several isolated points

GLfloat pt[2] = {3.0, 4.0};
glBegin(GL_POINTS);
glVertex2f(1.0, 2.0);    // x=1, y=2
glVertex2f(2.0, 3.0);    // x=2, y=3
glVertex2fv(pt);         // x=3, y=4
glVertex2i(4,5);         // x=4, y=5
glEnd();
```

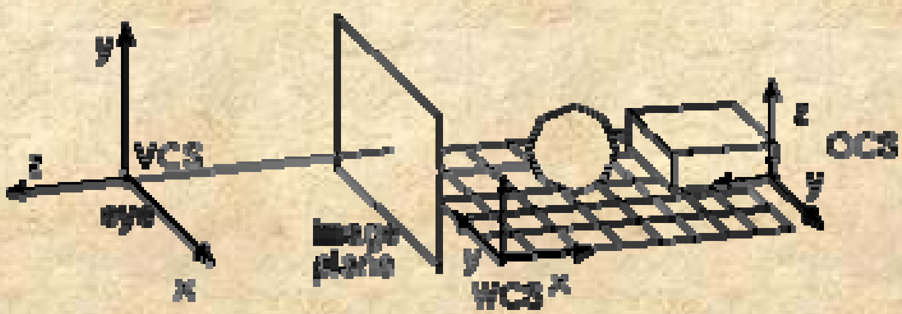
The following code fragment specifies a 3D polygon to be drawn, in this case a simple square. Note that in this case the same square could have been drawn using the GL_QUADS and GL_QUAD_STRIP primitives.

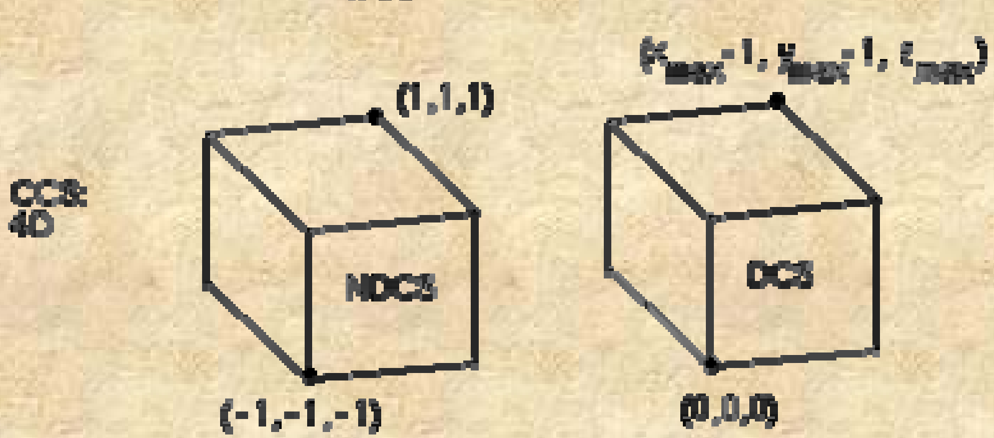
```
GLfloat p1[3] = {0,0,1};
GLfloat p2[3] = {1,0,1};
GLfloat p3[3] = {1,1,1};
GLfloat p4[3] = {0,1,1};

glBegin(GL_POLYGON);
glVertex3fv(p1);
glVertex3fv(p2);
glVertex3fv(p3);
glVertex3fv(p4);
glEnd();
```

Coordinate Systems in the Graphics Pipeline

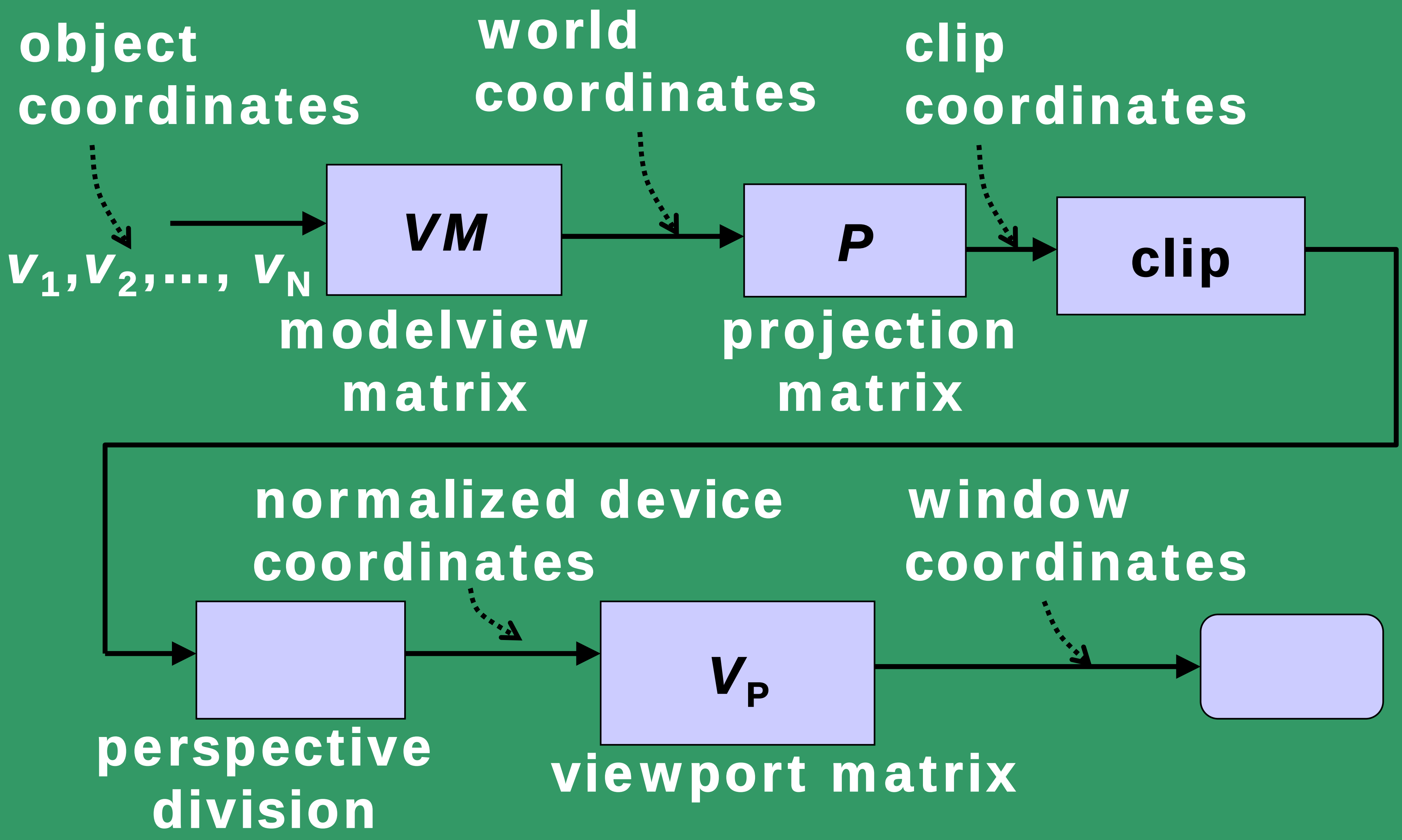
OCS - object coordinate system
WCS - world coordinate system
VCS - viewing coordinate system
CCS - clipping coordinate system
NDCS - normalized device coordinate system
DCS - device coordinate system





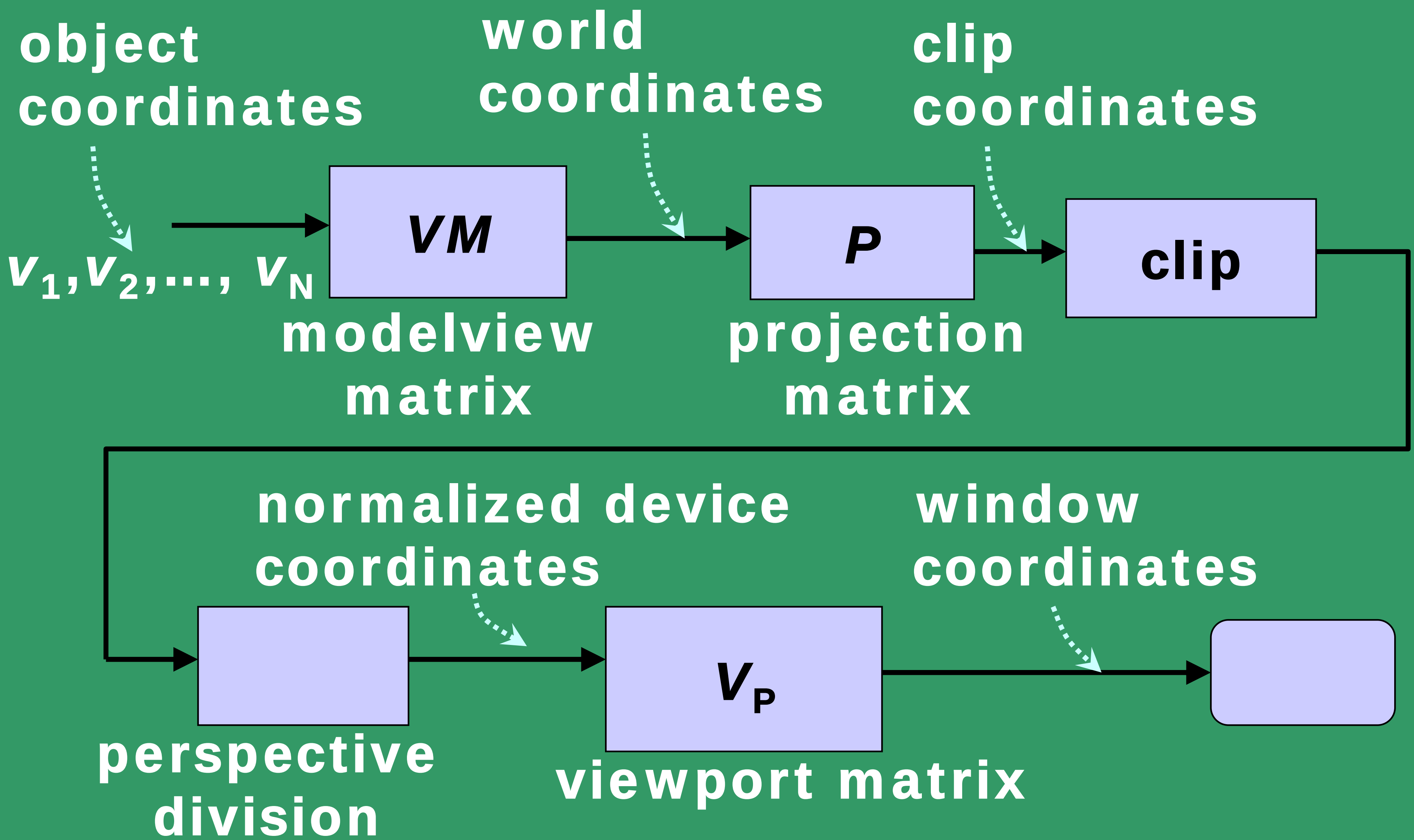


3D Viewing Pipeline



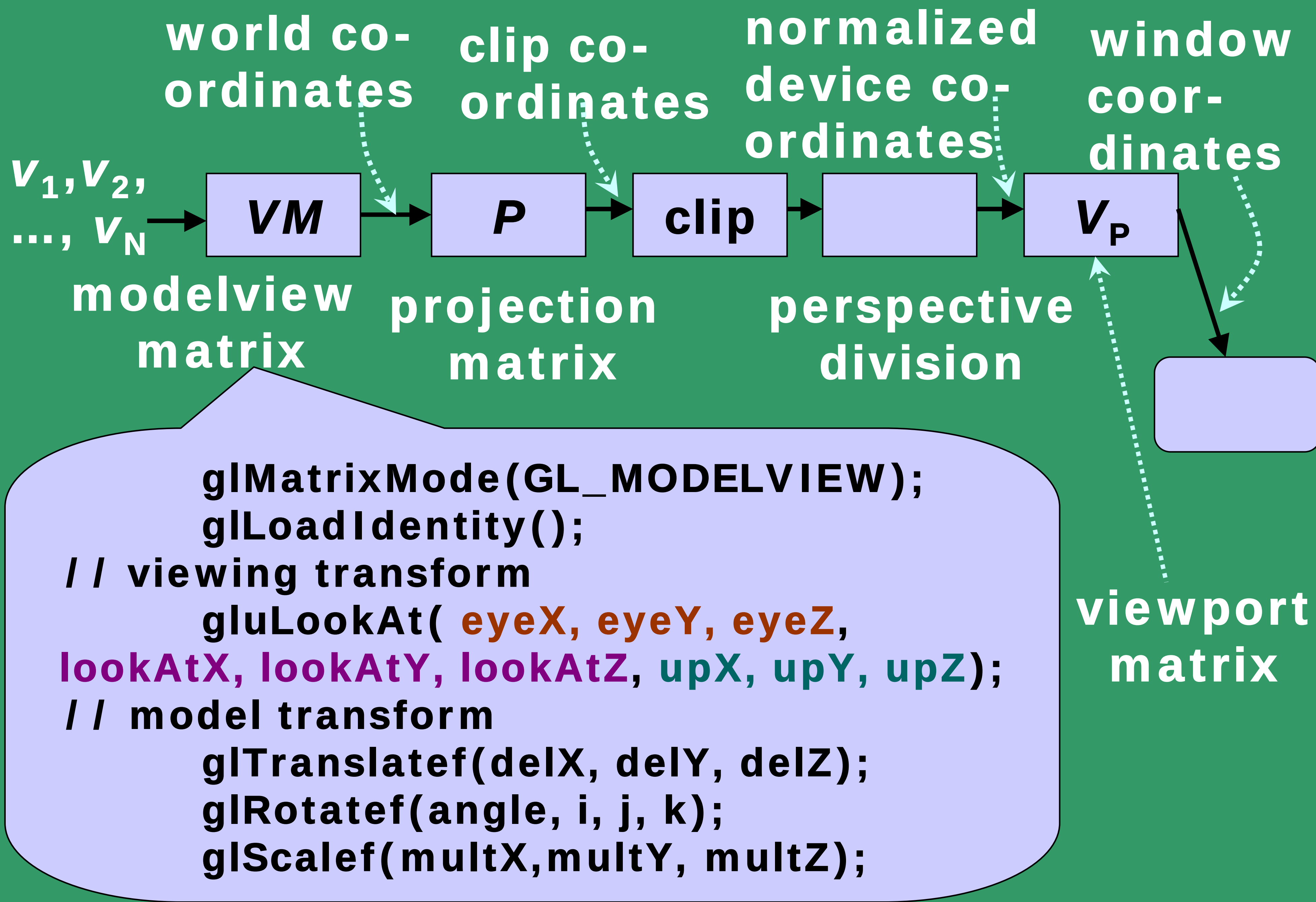
From F. S. Hill Jr., Computer Graphics using OpenGL

3D Viewing Pipeline

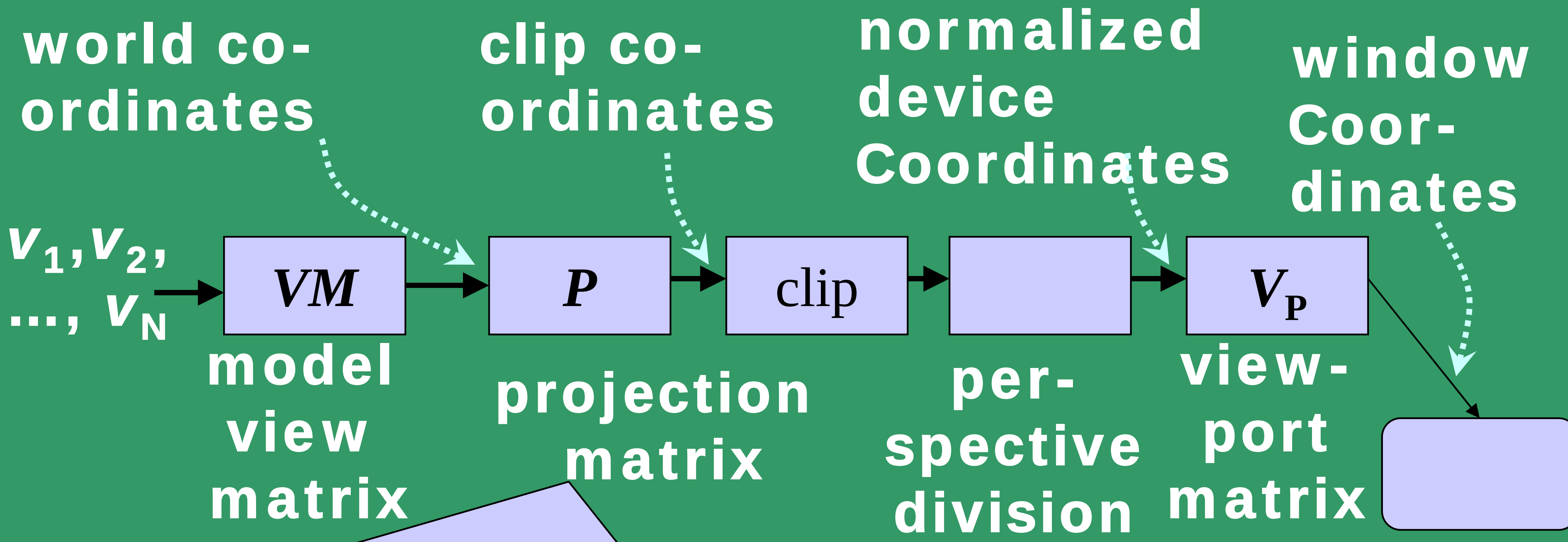


From F. S. Hill Jr., Computer Graphics using OpenGL

3D Viewing – ModelView Matrix



3D Viewing – Projection Matrix




```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
// perspective transform
gluPerspective( viewAngle, aspectRatio,nearZ,farZ );
// other commands for setting projection matrix
glFrustum(left, right, top, bottom);
glOrtho(left, right, top, bottom);
gluOrtho2D(left, right, top, bottom);
```

OpenGL functions for setting up transformations

modelling transformation
(modelview matrix)

glTranslatef()
glRotatef()
glScalef()

viewing transformation
(modelview matrix)

gluLookAt()

projection transformation
(projection matrix)

glFrustum()
gluPerspective()
glOrtho()
gluOrtho2D()

viewing transformation

glViewport()

Structure of a GLUT Program

```
int main(int argc, char **argv) {
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE |
    GLUT_RGB | GLUT_DEPTH);
    glutCreateWindow("Interactive rotating
    cube"); // with size & position
    glutDisplayFunc(display);
    // display callback, routines for drawing
    glutKeyboardFunc(myKeyHandler);
    // keyboard callback
    glutMouseFunc(myMouseClickHandler);
    // mouse callback
```

```
    glutMotionFunc(myMouseMotionHandler);
    // mouse move callback

    init();

    glutMainLoop();

}

void display() {...}

void myKeyHandler( unsigned char key, int x,
int y) {...}

void myMouseClickHandler( int button, int
state, int x, int y ) {...}

void myMouseMotionHandler( int x, int y) {...}
```

glutInitDisplaymode()

Before opening a graphics window, we need to decide on the `depth' of the buffers associated with the window. The following table shows the types of parameters that can be stored on a per-pixel basis:

The various GLUT_* options are invoked together by OR-ing them together, as illustrated in the example code, which creates a graphics window which has only a single copy of all buffers (GLUT_SINGLE), does not have an alpha buffer (GLUT_RGB), and has a depth buffer (GLUT_DEPTH).

RGB	Red, green and blue, Typically 8 bits per pixel	GLUT_RGB
A	Alpha or accumulation buffer, Used for composting images	GLUT_RGBA
Z	Depth value, used for Z-buffer visibility tests	GLUT_DEPTH
Double buffer	Extra copy of all buffers, Used for smoothing animation	GLUT_DOUBLE
Stencil buffer	Several extra bits, Useful in composting images	GLUT_STENCIL

glutInitWindowPosition(),   glutInitWindowSize(),   glutCreateWindow()

These calls assign an initial position, size, and name to the window and create the window itself.

glClearColor(),   glMatrixMode(),   glLoadIdentity(),   glOrtho()

glClearColor() sets the colour to be used when clearing the window. The remaining calls are used to define the type of camera projection. In this case, an orthographic projection is specified using a call to glOrtho(x1,x2,y1,y2,z1,z2). This defines the field of view of the camera, in this case 0<=x<=10, 0<=y<=10, -1<=z<=1.

glutDisplayFunc(display),   glutMainLoop()

This provides the name of the function you would like to have called whenever glut thinks the window needs to be redrawn. Thus, when the window is first created and whenever the window is uncovered or moved, the user-defined display() function will be called.

glutDisplayFunc() registers the call-back function, while glutMainLoop() hands execution control over to the glut library.

Viewing in 2D

```
void init(void) {

    glClearColor(0.0, 0.0, 0.0, 0.0);
    glColor3f(1.0f, 0.0f, 1.0f);
    glPointSize(1.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();

    gluOrtho2D(
        0.0, // left
        screenWidth, // right
        0.0, // bottom
        screenHeight); // top

}
```

Drawing in 2D

```
glBegin(GL_POINTS);
      glVertex2d(x1, y1);
      glVertex2d(x2, y2);
      .
      .
      .
      glVertex2d(xn, yn);
glEnd();
```

GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_POLYGON

Drawing a square in OpenGL

The following code fragment demonstrates a very simple OpenGL program which opens a graphics window and draws a square. It also prints 'helllo world' in the console window. The code is illustrative of the use of the glut library in opening the graphics window and managing the display loop.

glutInit()

Following the initial print statement, the glutInit() call initializes the GLUT library and also processes any command line options related to glut. These command line options are window-system dependent.

display()

The display() call-back function clears the screen, sets the current colour to red and draws a square polygon. The last call, glFlush(), forces previously issued OpenGL commands to begin execution.

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

void display(void)
{
 glClear( GL_COLOR_BUFFER_BIT);
 glColor3f(0.0, 1.0, 0.0);
 glBegin(GL_POLYGON);
  glVertex3f(2.0, 4.0, 0.0);
  glVertex3f(8.0, 4.0, 0.0);
  glVertex3f(8.0, 6.0, 0.0);
  glVertex3f(2.0, 6.0, 0.0);
 glEnd();
 glFlush();
}

int main(int argc, char **argv)
{
 printf("hello world\n");
 glutInit(&argc, argv);
 glutInitDisplayMode
       ( GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);

 glutInitWindowPosition(100,100);
 glutInitWindowSize(300,300);
 glutCreateWindow ("square");

 glClearColor(0.0, 0.0, 0.0, 0.0);
        // black background
 glMatrixMode(GL_PROJECTION);
       // setup  viewing projection
 glLoadIdentity();
        // start with identity matrix
 glOrtho(0.0, 10.0, 0.0, 10.0, -1.0, 1.0);
       // setup a 10x10x2 viewing world

 glutDisplayFunc(display);
 glutMainLoop();

 return 0;
}
```

Assigning Colours

OpenGL maintains a current drawing colour as part of its state information.

The glColor() function calls are used to change the current drawing colour - assigned using the glColor function call.

Like glVertex(), this function exists in various instantiations. Colour components are specified in the order of red, green, blue. Colour component values are in the range [0...1], where 1 corresponds to maximum intensity.

For unsigned bytes, the range corresponds to [0...255]. All primitives following the fragment of code given below would be drawn in green, assuming no additional glColor() function calls are used.

Color Flashing

Applications that use colors deal with them in one of two ways:

- **RGB, also called TrueColor -- Every pixel has a red, green, and a blue value associated with it.**
- **via a Color LookUp Table (CLUT), also called color index mode -- Every pixel has a color index associated with it. The color index is a pointer into the color lookup table where the real RGB values reside.**

The use of a color lookup table takes significantly less memory but provides for fewer colors. Most 3D applications, and OpenGL in particular, operate using RGB colors because it is the natural color space for colors and lighting and shading. Color flashing will occur when you run OpenGL. When the focus shifts to an OpenGL window, either by clicking on it or by moving the mouse pointer to it, the way you have instructed X to change focus, the colors of the rest of the windows will change dramatically. When a non-OpenGL window is in focus, the colors in the OpenGL window will change.

Assigning Colours

Current drawing colour maintained as a state.

Colour components - red, green, blue in range [0...1] as float or [0...255] as unsigned byte

```
GLfloat myColour[3] = {0, 0, 1}; // blue

glColor3fv( myColour ); // using vector of floats

glColor3f(1.0, 0.0, 0.0); // red using floats

glColor3ub(0, 255, 0);   // green using unsigned bytes
```

Colour Interpolation

If desired, a polygon can be smoothly shaded to interpolate colours between vertices.

This is accomplished by using the GL_SMOOTH shading mode (the OpenGL default) and by assigning a desired colour to each vertex.

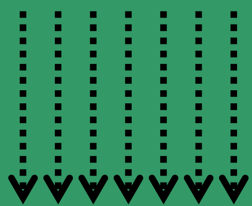
```
glShadeModel(GL_SMOOTH);
// as opposed to GL_FLAT

glBegin(GL_POLYGON);
        glColor3f(1.0, 0, 0 ); // red
        glVertex2d(0, 0);
        glColor3f(0, 0, 1.0 ); // blue
        glVertex2d(1, 0);
        glColor3f(0, 1.0, 0 ); // green
        glVertex2d(1, 1);
        glColor3f(1.0, 1.0, 1.0 ); // white
        glVertex2d(0, 1);
glEnd();
```

A fourth value called alpha is often appended to the colour vector. This can be used assign a desired level of transparency to a primitive and finds uses in compositing multiple images together. An alpha value of 0.0 defines an opaque colour, while an alpha value of 1.0 corresponds to complete transparency.

The screen can be cleared to a particular colour as follows:

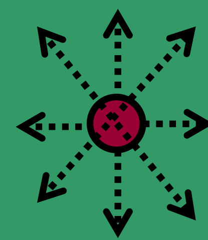
```
glClearcolor(1.0, 1.0, 1.0, 0.0);          // sets the clear colour to
                                           white and opaque

glClear( GL_COLOR_BUFFER_BIT);             // clears the colour
                                           frame buffer
```

Lighting up the 3D World



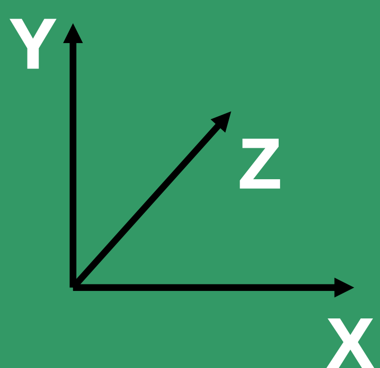
Ambient light

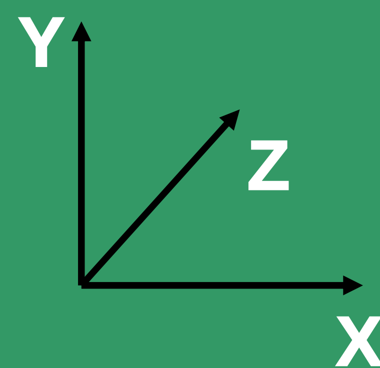
(source at infinity)



Diffuse light

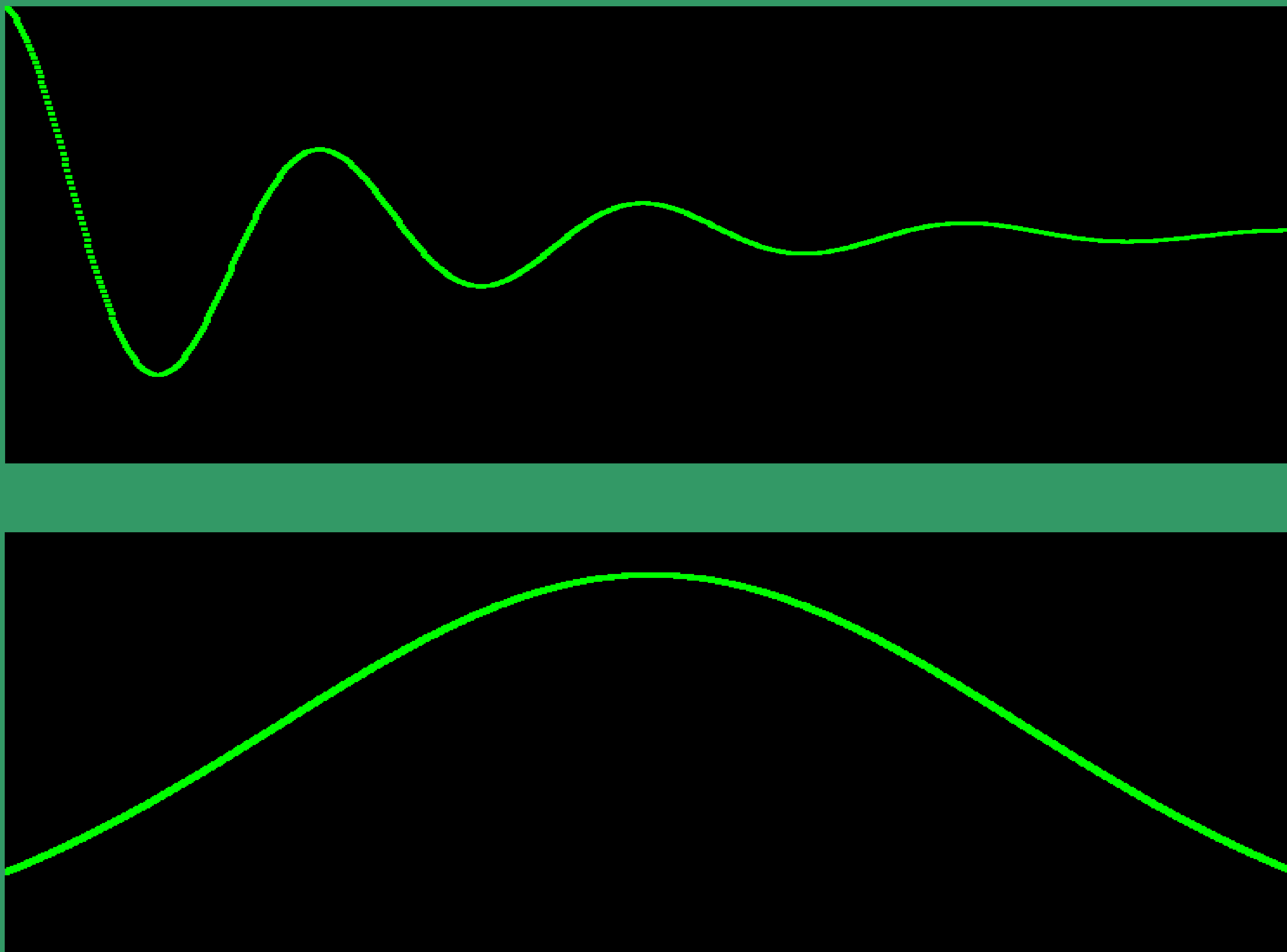
(from a point source)





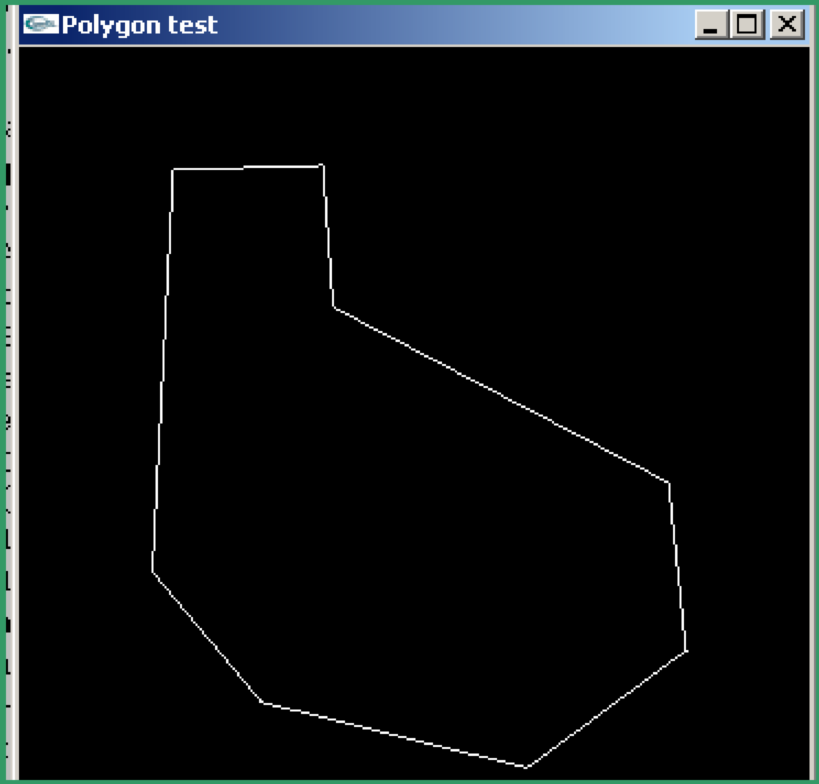
```
GLfloat light0_colour[] = {1, 1.0, 1, 1.0};
GLfloat light0_position[] = {0.0, 1.0, 0.0, 0.0};

// Setting up light type and position
glLightfv(GL_LIGHT0, GL_AMBIENT,
light0_colour); // use GL_DIFFUSE for diffuse

glLightfv(GL_LIGHT0, GL_POSITION,
light0_position);


//  Enable light
glEnable(GL_LIGHT0); // can have other lights
glEnable(GL_LIGHTING);
glShadeModel(GL_SMOOTH);
```

Demo – 2D Curves

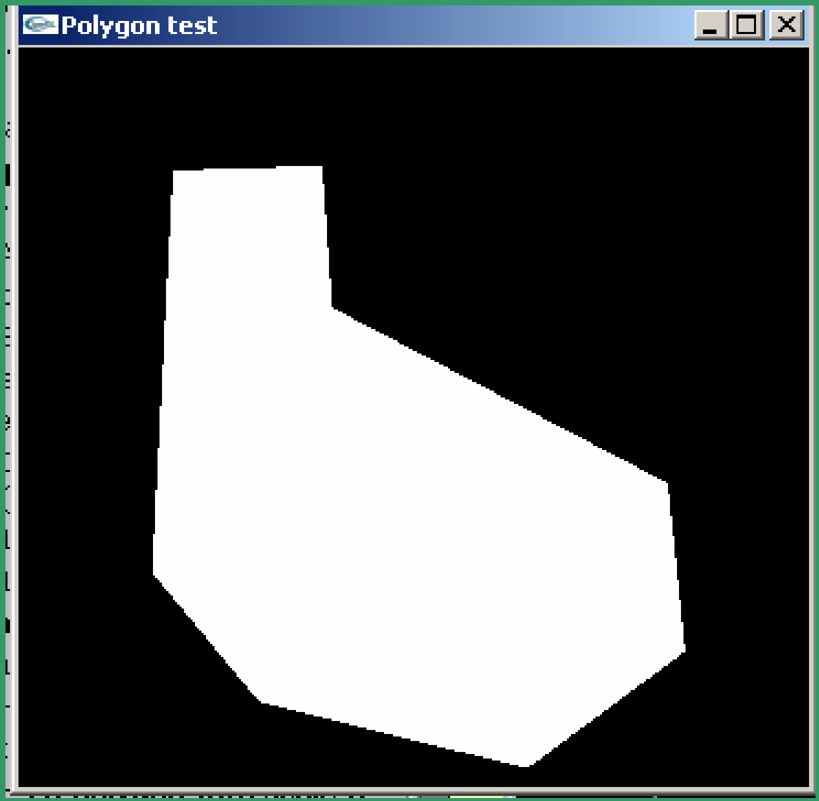


Demo – 2D Polygon Drawing

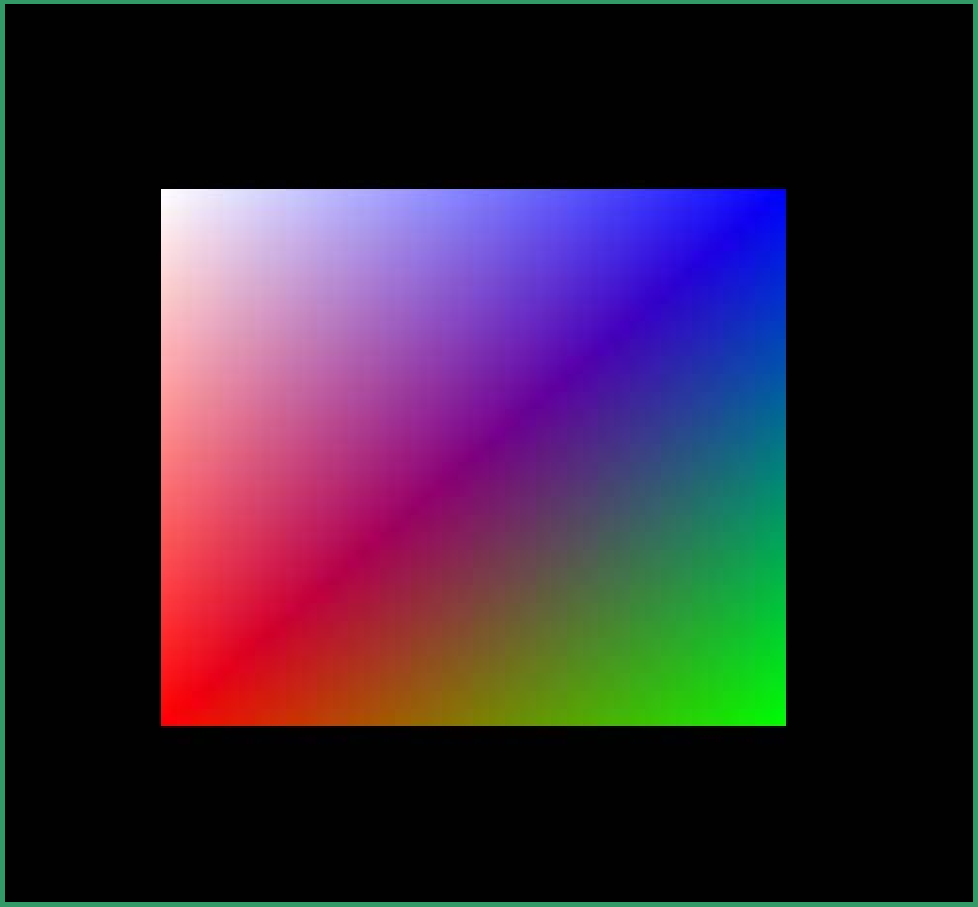
Polyline test



Polygon test



Demo – Colour Interpolation



References

- OpenGL Architecture Review Board, Dave Shreiner, Mason Woo, Jackie Neider, Tom Davis, OpenGL Architecture Review Board, The OpenGL Programming Guide – The Red book, 4th edition, Addison-Wesley.

  (http://www.glprogramming.com/red/index.html)

- OpenGL Architecture Review Board, Dave Shreiner, The OpenGL Reference Manual- The Blue book, 4th edition, Addison-Wesley.

  (http://rush3d.com/reference/opengl-bluebook-1.0)

- F. S. Hill Jr., Computer Graphics using OpenGL, Pearson Education, 2003.

End of Lectures on Graphics Programming using OpenGL