

UNIT III - GRAPHICS PROGRAMMING

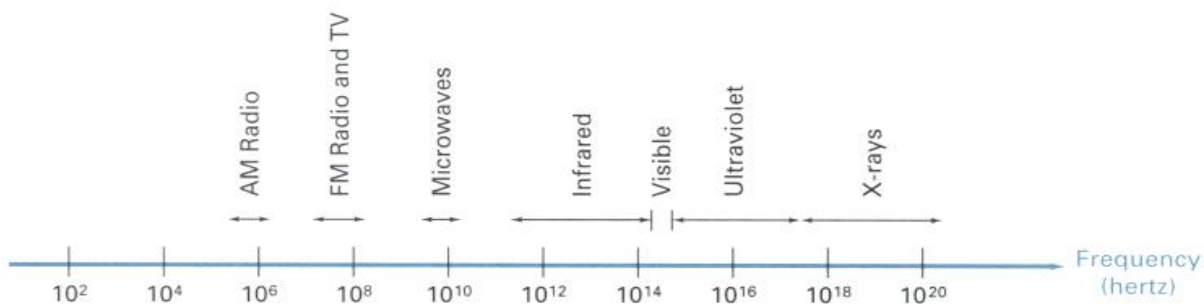
Color Models – RGB, YIQ, CMY, HSV – Animations – General Computer Animation, Raster, Keyframe - Graphics programming using OpenGL – Basic graphics primitives – Drawing three dimensional objects - Drawing three dimensional scenes

Color Models

Color Model is a method for explaining the properties or behavior of color within some particular context. No single color model can explain all aspects of color, so we make use of different models to help describe the different perceived characteristics of color.

Properties of Light

- Light is a narrow frequency band within the electromagnetic system.
- Other frequency bands within this spectrum are called radio waves, micro waves, infrared waves and x-rays. The below fig shows the frequency ranges for some of the electromagnetic bands.



- Each frequency value within the visible band corresponds to a distinct color.
- At the low frequency end is a red color (4.3×10^{14} Hz) and the highest frequency is a violet color (7.5×10^{14} Hz)
- Spectral colors range from the reds through orange and yellow at the low frequency end to greens, blues and violet at the high end.
- Since light is an electro magnetic wave, the various colors are described in terms of either the frequency for the wave length λ of the wave.
- The wave length and frequency of the monochromatic wave are inversely proportional to each other, with the proportionality constants as the speed of light C where $C = \lambda f$

- A light source such as the sun or a light bulb emits all frequencies within the visible range to produce white light. When white light is incident upon an object, some frequencies are reflected and some are absorbed by the object. The combination of frequencies present in the reflected light determines what we perceive as the color of the object.
- If low frequencies are predominant in the reflected light, the object is described as red. In this case, the perceived light has the dominant frequency at the red end of the spectrum. The dominant frequency is also called the hue, or simply the color of the light.
- Brightness is another property, which in the perceived intensity of the light.
- Intensity in the radiant energy emitted per unit time, per unit solid angle, and per unit projected area of the source.
- Radiant energy is related to the luminance of the source.
- The next property in the purity or saturation of the light.
 - Purity describes how washed out or how pure the color of the light appears.
 - Pastels and Pale colors are described as less pure.
- The term chromaticity is used to refer collectively to the two properties, purity and dominant frequency.
- Two different color light sources with suitably chosen intensities can be used to produce a range of other colors.
- If the 2 color sources combine to produce white light, they are called complementary colors. E.g., Red and Cyan, green and magenta, and blue and yellow.
- Color models that are used to describe combinations of light in terms of dominant frequency use 3 colors to obtain a wide range of colors, called the color gamut.
- The 2 or 3 colors used to produce other colors in a color model are called primary colors.

Standard Primaries

XYZ Color Model

- The set of primaries is generally referred to as the XYZ or (X,Y,Z) color model where X,Y and Z represent vectors in a 3D, additive color space.
- Any color $C\lambda$ is expressed as

$$C\lambda = XX + YY + ZZ \quad \text{-----}(1)$$

Where X,Y and Z designates the amounts of the standard primaries needed to match $C\lambda$.

- It is convenient to normalize the amount in equation (1) against luminance (X + Y + Z). Normalized amounts are calculated as,

$$x = X/(X+Y+Z), \quad y = Y/(X+Y+Z), \quad z = Z/(X+Y+Z)$$

with $x + y + z = 1$

- Any color can be represented with just the x and y amounts. The parameters x and y are called the chromaticity values because they depend only on hue and purity.
- If we specify colors only with x and y, we cannot obtain the amounts X, Y and Z. so, a complete description of a color is given with the 3 values x, y and Y.

$$X = (x/y)Y, \quad Z = (z/y)Y$$

Where $z = 1-x-y$.

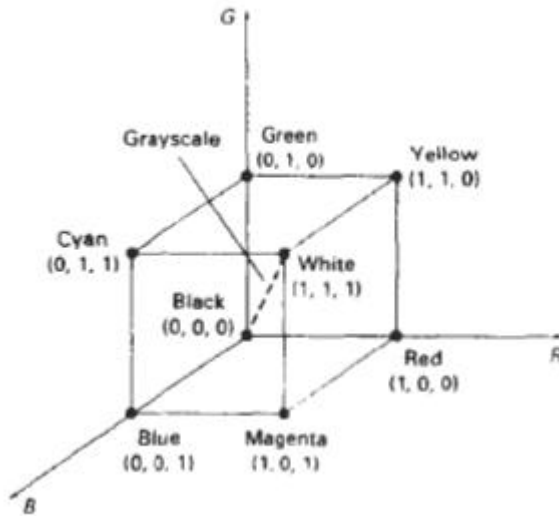
Intuitive Color Concepts

- Color paintings can be created by mixing color pigments with white and black pigments to form the various shades, tints and tones.
- Starting with the pigment for a 'pure color' the color is added to black pigment to produce different shades. The more black pigment produces darker shades.
- Different tints of the color are obtained by adding a white pigment to the original color, making it lighter as more white is added.
- Tones of the color are produced by adding both black and white pigments.

RGB Color Model

- Based on the tristimulus theory of vision, our eyes perceive color through the stimulation of three visual pigments in the cones on the retina.

- These visual pigments have a peak sensitivity at wavelengths of about 630 nm (red), 530 nm (green) and 450 nm (blue).
- By comparing intensities in a light source, we perceive the color of the light.
- This is the basis for displaying color output on a video monitor using the 3 color primaries, red, green, and blue referred to as the RGB color model. It is represented in the below figure.



- The sign represents black, and the vertex with coordinates (1,1,1) in white.
- Vertices of the cube on the axes represent the primary colors, the remaining vertices represents the complementary color for each of the primary colors.
- The RGB color scheme is an additive model. (i.e.,) Intensities of the primary colors are added to produce other colors.
- Each color point within the bounds of the cube can be represented as the triple (R,G,B) where values for R, G and B are assigned in the range from 0 to 1.
- The color $C\lambda$ is expressed in RGB component as

$$C\lambda = R\mathbf{R} + G\mathbf{G} + B\mathbf{B}$$

- The magenta vertex is obtained by adding red and blue to produce the triple (1,0,1) and white at (1,1,1) in the sum of the red, green and blue vertices.
- Shades of gray are represented along the main diagonal of the cube from the origin (black) to the white vertex.

2.5.5 YIQ Color Model

- The National Television System Committee (NTSC) color model for forming the composite video signal in the YIQ model.
- In the YIQ color model, luminance (brightness) information is contained in the Y parameter, chromaticity information (hue and purity) is contained into the I and Q parameters.
- A combination of red, green and blue intensities are chosen for the Y parameter to yield the standard luminosity curve.
- Since Y contains the luminance information, black and white TV monitors use only the Y signal.
- Parameter I contains orange-cyan hue information that provides the flesh-tone shading and occupies a bandwidth of 1.5 MHz.
- Parameter Q carries green-magenta hue information in a bandwidth of about 0.6 MHz.
- An RGB signal can be converted to a TV signal using an NTSC encoder which converts RGB values to YIQ values, as follows

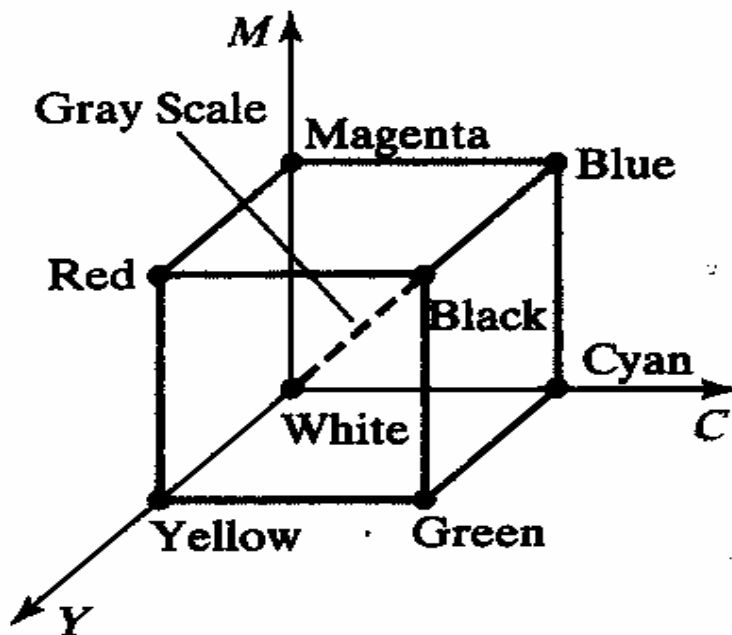
$$\begin{bmatrix} Y \\ I \\ Q \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.144 \\ 0.596 & -0.275 & -0.321 \\ 0.212 & -0.528 & 0.311 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

- An NTSC video signal can be converted to an RGB signal using an NTSC decoder which separates the video signal into YIQ components, then converts to RGB values, as follows:

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1.000 & 0.956 & 0.620 \\ 1.000 & -0.272 & -0.647 \\ 1.000 & -1.108 & 1.705 \end{bmatrix} \begin{bmatrix} Y \\ I \\ Q \end{bmatrix}$$

CMY Color Model

- A color model defined with the primary colors cyan, magenta, and yellow (CMY) is useful for describing color output to hard copy devices.
- It is a subtractive color model (i.e.,) cyan can be formed by adding green and blue light. When white light is reflected from cyan-colored ink, the reflected light must have no red component. i.e., red light is absorbed or subtracted by the ink.
- Magenta ink subtracts the green component from incident light and yellow subtracts the blue component.



- In CMY model, point (1,1,1) represents black because all components of the incident light are subtracted.
- The origin represents white light.
- Equal amounts of each of the primary colors produce grays along the main diagonal of the cube.
- A combination of cyan and magenta ink produces blue light because the red and green components of the incident light are absorbed.
- The printing process often used with the CMY model generates a color point with a collection of 4 ink dots; one dot is used for each of the primary colors (cyan, magenta and yellow) and one dot in black.

- The conversion from an RGB representation to a CMY representation is expressed as

$$\begin{bmatrix} C \\ M \\ Y \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

Where the white is represented in the RGB system as the unit column vector.

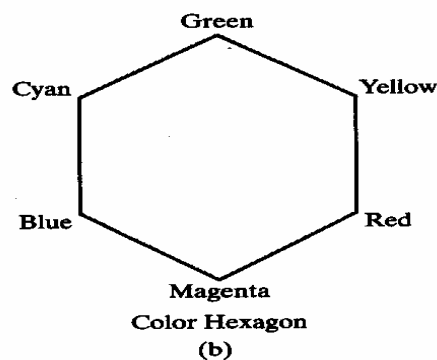
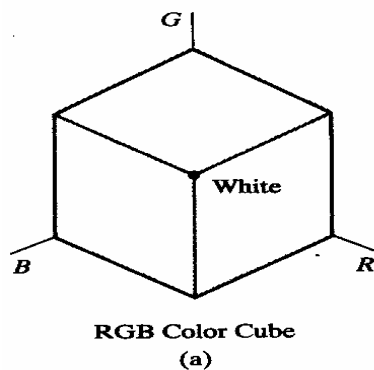
- Similarly the conversion of CMY to RGB representation is expressed as

$$\begin{bmatrix} R \\ G \\ B \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix} - \begin{bmatrix} C \\ M \\ Y \end{bmatrix}$$

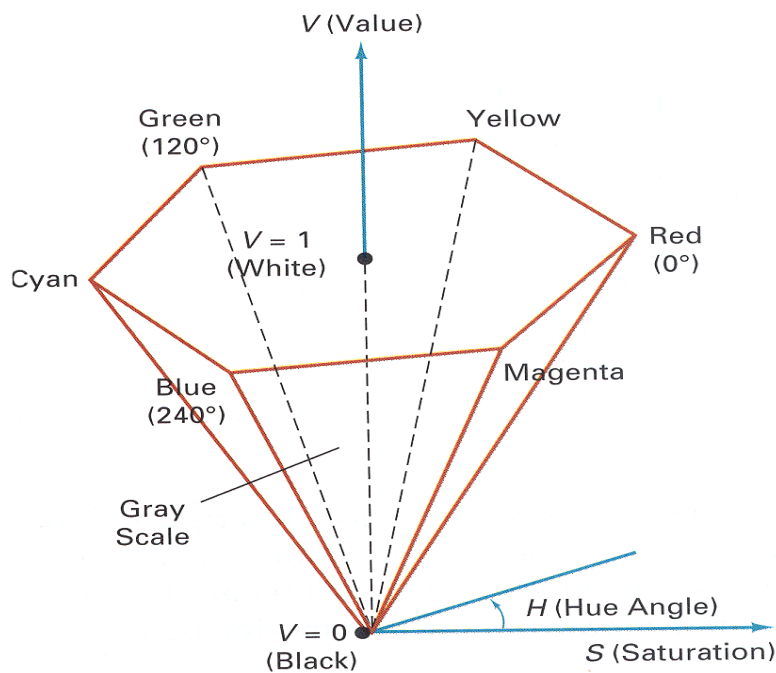
Where black is represented in the CMY system as the unit column vector.

HSV Color Model

- The HSV model uses color descriptions that have a more interactive appeal to a user.
- Color parameters in this model are hue (H), saturation (S), and value (V).
- The 3D representation of the HSV model is derived from the RGB cube. The outline of the cube has the hexagon shape.



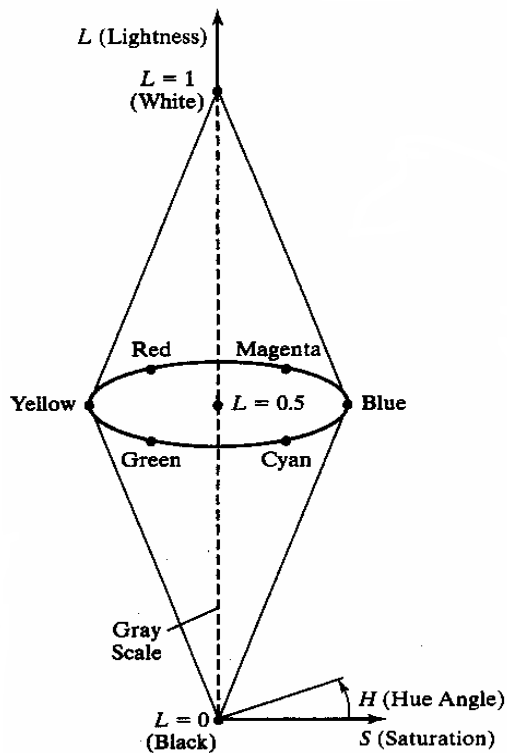
- The boundary of the hexagon represents the various hues, and it is used as the top of the HSV hexcone.
- In the hexcone, saturation is measured along a horizontal axis, and value is along a vertical axis through the center of the hexcone.
- Hue is represented as an angle about the vertical axis, ranging from 0° at red through 360° . Vertices of the hexagon are separated by 60° intervals.
- Yellow is at 60° , green at 120° and cyan opposite red at $H = 180^\circ$. Complementary colors are 180° apart.



- Saturation S varies from 0 to 1. the maximum purity at $S = 1$, at $S = 0.25$, the hue is said to be one quarter pure, at $S = 0$, we have the gray scale.
- Value V varies from 0 at the apex to 1 at the top.
 - the apex representation black.
- At the top of the hexcone, colors have their maximum intensity.
- When $V = 1$ and $S = 1$ we have the 'pure' hues.
- White is the point at $V = 1$ and $S = 0$.

HLS Color Model

- HLS model is based on intuitive color parameters used by Tektronix.
- It has the double cone representation shown in the below figure. The 3 parameters in this model are called Hue (H), lightness (L) and saturation (s).



- Hue specifies an angle about the vertical axis that locates a chosen hue.
- In this model $H = 0^\circ$ corresponds to Blue.
- The remaining colors are specified around the perimeter of the cone in the same order as in the HSV model.
- Magenta is at 60° , Red in at 120° , and cyan in at $H = 180^\circ$.
- The vertical axis is called lightness (L). At $L = 0$, we have black, and white is at $L = 1$. Gray scale is along the L axis and the “pure hues” on the $L = 0.5$ plane.
- Saturation parameter S specifies relative purity of a color. S varies from 0 to 1. Pure hues are those for which $S = 1$ and $L = 0.5$.

- As S decreases, the hues are said to be less pure.
- At $S=0$, it is said to be gray scale.

Animation

- Computer animation refers to any time sequence of visual changes in a scene.
- Computer animations can also be generated by changing camera parameters such as position, orientation and focal length.
- Applications of computer-generated animation are entertainment, advertising, training and education.

Example : Advertising animations often transition one object shape into another.

Frame-by-Frame animation

Each frame of the scene is separately generated and stored. Later, the frames can be recoded on film or they can be consecutively displayed in "real-time playback" mode

Design of Animation Sequences

An animation sequence is designed with the following steps:

- Story board layout
- Object definitions
- Key-frame specifications
- Generation of in-between frames.

Story board

- The story board is an outline of the action.
- It defines the motion sequences as a set of basic events that are to take place.
- Depending on the type of animation to be produced, the story board could consist of a set of rough sketches or a list of the basic ideas for the motion.

Object Definition

- An object definition is given for each participant in the action.

- Objects can be defined in terms of basic shapes such as polygons or splines.
- The associated movements of each object are specified along with the shape.

Key frame

- A key frame is detailed drawing of the scene at a certain time in the animation sequence.
- Within each key frame, each object is positioned according to the time for that frame.
- Some key frames are chosen at extreme positions in the action; others are spaced so that the time interval between key frames is not too much.

In-betweens

- In betweens are the intermediate frames between the key frames.
- The number of in between needed is determined by the media to be used to display the animation.
- Film requires 24 frames per second and graphics terminals are refreshed at the rate of 30 to 60 frames per seconds.
- Time intervals for the motion are setup so there are from 3 to 5 in-between for each pair of key frames.
- Depending on the speed of the motion, some key frames can be duplicated.
- For a 1 min film sequence with no duplication, 1440 frames are needed.
- Other required tasks are
 - Motion verification
 - Editing
 - Production and synchronization of a sound track.

General Computer Animation Functions

Steps in the development of an animation sequence are,

1. Object manipulation and rendering

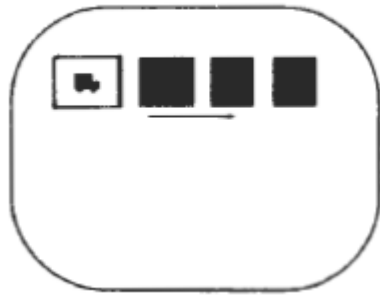
2. Camera motion

3. Generation of in-betweens

- Animation packages such as wave front provide special functions for designing the animation and processing individuals objects.
- Animation packages facilitate to store and manage the object database.
- Object shapes and associated parameter are stored and updated in the database.
- Motion can be generated according to specified constraints using 2D and 3D transformations.
- Standard functions can be applied to identify visible surfaces and apply the rendering algorithms.
- Camera movement functions such as zooming, panning and tilting are used for motion simulation.
- Given the specification for the key frames, the in-betweens can be automatically generated.

Raster Animations

- On raster systems, real-time animation in limited applications can be generated using raster operations.
- Sequence of raster operations can be executed to produce real time animation of either 2D or 3D objects.
- We can animate objects along 2D motion paths using the color-table transformations.
 - Predefine the object as successive positions along the motion path, set the successive blocks of pixel values to color table entries.
 - Set the pixels at the first position of the object to 'on' values, and set the pixels at the other object positions to the background color.
 - The animation is accomplished by changing the color table values so that the object is 'on' at successive positions along the animation path as the preceding position is set to the background intensity.



Computer Animation Languages

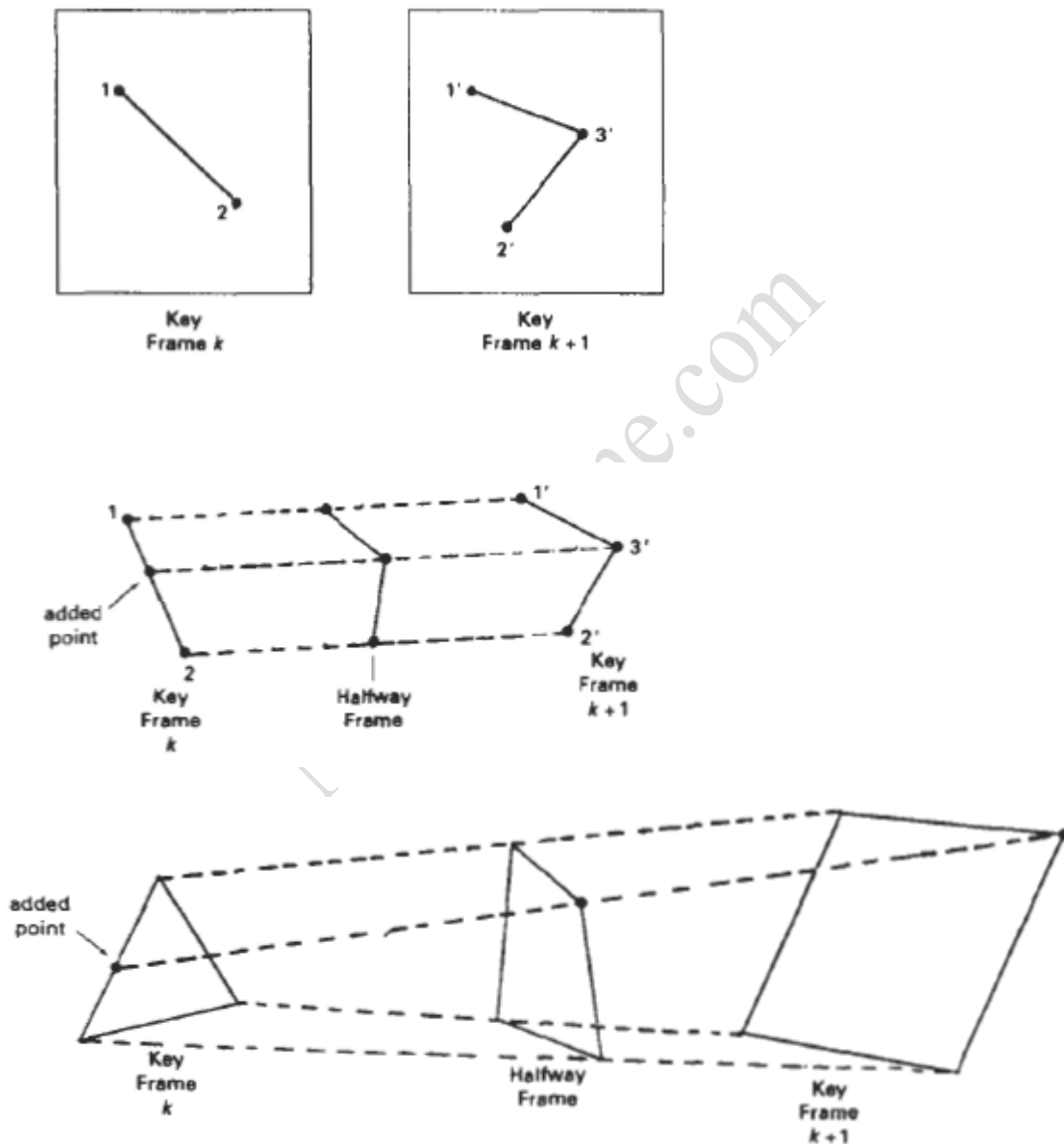
- Animation functions include a graphics editor, a key frame generator and standard graphics routines.
- The graphics editor allows designing and modifying object shapes, using spline surfaces, constructive solid geometry methods or other representation schemes.
- Scene description includes the positioning of objects and light sources defining the photometric parameters and setting the camera parameters.
- Action specification involves the layout of motion paths for the objects and camera.
- Keyframe systems are specialized animation languages designed dimply to generate the in-betweens from the user specified keyframes.
- Parameterized systems allow object motion characteristics to be specified as part of the object definitions. The adjustable parameters control such object characteristics as degrees of freedom motion limitations and allowable shape changes.
- Scripting systems allow object specifications and animation sequences to be defined with a user input script. From the script, a library of various objects and motions can be constructed.

Keyframe Systems

- Each set of in-betweens are generated from the specification of two keyframes.
- For complex scenes, we can separate the frames into individual components or objects called cells, an acronym from cartoon animation.

Morphing

- Transformation of object shapes from one form to another is called Morphing.
- Morphing methods can be applied to any motion or transition involving a change in shape. The example is shown in the below figure.



- The general preprocessing rules for equalizing keyframes in terms of either the number of vertices to be added to a keyframe.

- Suppose we equalize the edge count and parameters L_k and L_{k+1} denote the number of line segments in two consecutive frames. We define,

$$L_{\max} = \max (L_k, L_{k+1})$$

$$L_{\min} = \min(L_k, L_{k+1})$$

$$N_e = L_{\max} \bmod L_{\min}$$

$$N_s = \text{int} (L_{\max}/L_{\min})$$

- The preprocessing is accomplished by
 1. Dividing N_e edges of keyframe_{\min} into N_s+1 section.
 2. Dividing the remaining lines of keyframe_{\min} into N_s sections.
- For example, if $L_k = 15$ and $L_{k+1} = 11$, we divide 4 lines of keyframe_{k+1} into 2 sections each. The remaining lines of keyframe_{k+1} are left intact.
- If the vector counts in equalized parameters V_k and V_{k+1} are used to denote the number of vertices in the two consecutive frames. In this case we define

$$V_{\max} = \max(V_k, V_{k+1}), V_{\min} = \min(V_k, V_{k+1}) \quad \text{and}$$

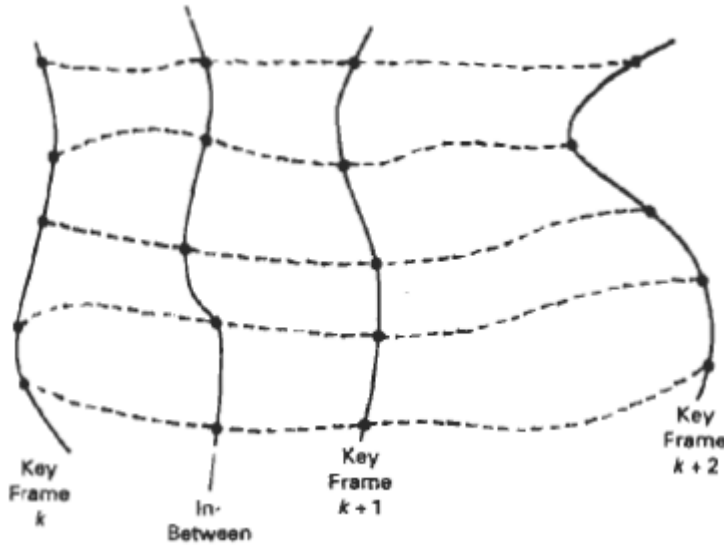
$$N_{ls} = (V_{\max} - 1) \bmod (V_{\min} - 1)$$

$$N_p = \text{int} ((V_{\max} - 1)/(V_{\min} - 1))$$

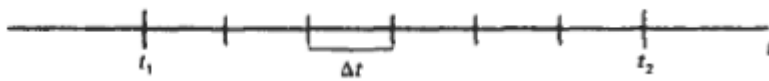
- Preprocessing using vertex count is performed by
 1. Adding N_p points to N_{ls} line section of keyframe_{\min} .
 2. Adding N_p-1 points to the remaining edges of keyframe_{\min} .

Simulating Accelerations

Curve-fitting techniques are often used to specify the animation paths between key frames. Given the vertex positions at the key frames, we can fit the positions with linear or nonlinear paths. Figure illustrates a nonlinear fit of key-frame positions. This determines the trajectories for the in-betweens. To simulate accelerations, we can adjust the time spacing for the in-betweens.



For constant speed (zero acceleration), we use equal-interval time spacing for the in-betweens. Suppose we want n in-betweens for key frames at times t_1 and t_2 .



The time interval between key frames is then divided into $n + 1$ subintervals, yielding an in-between spacing of

$$\Delta = (t_2 - t_1) / (n + 1)$$

we can calculate the time for any in-between as

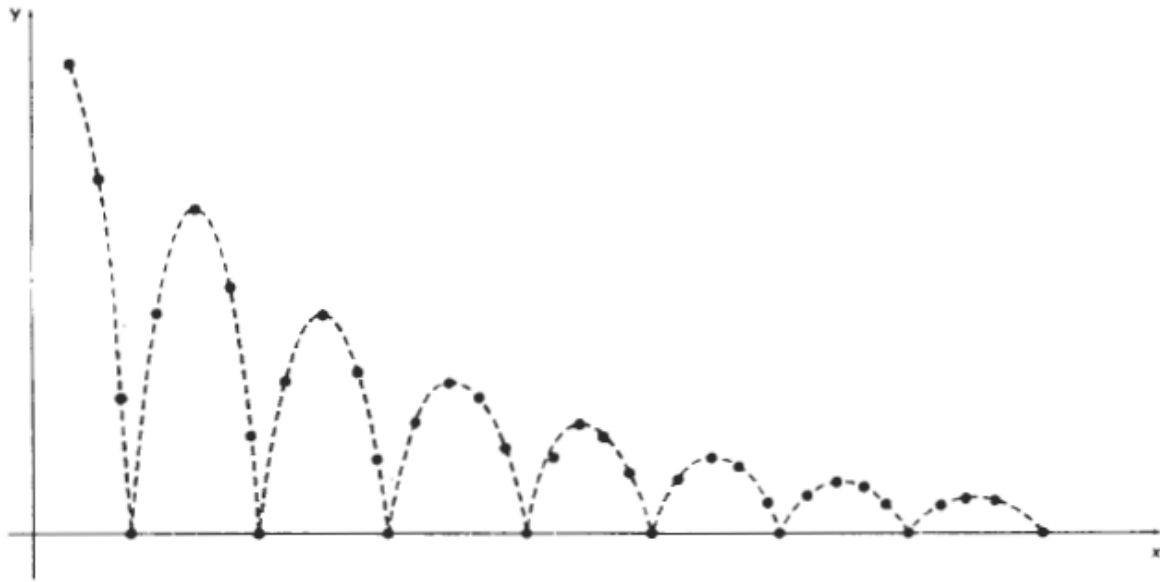
$$t_{Bj} = t_1 + j \Delta t, \quad j = 1, 2, \dots, n$$

Motion Specification

These are several ways in which the motions of objects can be specified in an animation system.

Direct Motion Specification

- Here the rotation angles and translation vectors are explicitly given.
- Then the geometric transformation matrices are applied to transform coordinate positions.



- We can approximate the path of a bouncing ball with a damped, rectified, sine curve

$$y(x) = A / \sin(\omega_x + \theta_0) / e^{-kx}$$

where A is the initial amplitude, ω is the angular frequency, θ_0 is the phase angle and k is the damping constant.

Goal Directed Systems

- We can specify the motions that are to take place in general terms that abstractly describe the actions.
- These systems are called goal directed. Because they determine specific motion parameters given the goals of the animation.
- Eg., To specify an object to 'walk' or to 'run' to a particular distance.

Kinematics and Dynamics

- With a kinematics description, we specify the animation by motion parameters (position, velocity and acceleration) without reference to the forces that cause the motion.
- For constant velocity (zero acceleration) we designate the motions of rigid bodies in a scene by giving an initial position and velocity vector for each object.

- We can specify accelerations (rate of change of velocity), speed up, slow downs and curved motion paths.
- An alternative approach is to use inverse kinematics; where the initial and final positions of the object are specified at specified times and the motion parameters are computed by the system.

Graphics programming using OPENGL

OpenGL is a software interface that allows you to access the graphics hardware without taking care of the hardware details or which graphics adapter is in the system. OpenGL is a low-level graphics library specification. It makes available to the programmer a small set of geometric primitives - points, lines, polygons, images, and bitmaps. OpenGL provides a set of commands that allow the specification of geometric objects in two or three dimensions, using the provided primitives, together with commands that control how these objects are rendered (drawn).

Libraries

- OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL commands to perform such tasks as setting up matrices for specific viewing orientations and projections and rendering surfaces.
- OpenGL Utility Toolkit (GLUT) is a window-system-independent toolkit, written by Mark Kilgard, to hide the complexities of differing window APIs.

Include Files

For all OpenGL applications, you want to include the gl.h header file in every file. Almost all OpenGL applications use GLU, the aforementioned OpenGL Utility Library, which also requires inclusion of the glu.h header file. So almost every OpenGL source file begins with:

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

If you are using the OpenGL Utility Toolkit (GLUT) for managing your window manager tasks, you should include:

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

The following files must be placed in the proper folder to run a OpenGL Program.

Libraries (place in the lib\ subdirectory of Visual C++)

- [opengl32.lib](#)
- [glu32.lib](#)
- [glut32.lib](#)

Include files (place in the include\GL\ subdirectory of Visual C++)

- [gl.h](#)
- [glu.h](#)
- [glut.h](#)

Dynamically-linked libraries (place in the \Windows\System subdirectory)

- [opengl32.dll](#)
- [glu32.dll](#)
- [glut32.dll](#)

Working with OpenGL

Opening a window for Drawing

The First task in making pictures is to open a screen window for drawing. The following five functions initialize and display the screen window in our program.

1. glutInit(&argc, argv)

The first thing we need to do is call the glutInit() procedure. It should be called before any other GLUT routine because it initializes the GLUT library. The parameters to glutInit() should be the same as those to main(), specifically main(int argc, char** argv) and glutInit(&argc, argv).

2. glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB)

The next thing we need to do is call the glutInitDisplayMode() procedure to specify the display mode for a window.

We must first decide whether we want to use an RGBA (GLUT_RGB) or color-index (GLUT_INDEX) color model. The RGBA mode stores its color buffers as red, green, blue, and alpha color components. Color-index mode, in contrast, stores color buffers in indicies. And for special effects, such as shading, lighting, and fog, RGBA mode provides more flexibility. In general, use RGBA mode whenever possible. RGBA mode is the default.

Another decision we need to make when setting up the display mode is whether we want to use single buffering (GLUT_SINGLE) or double buffering (GLUT_DOUBLE). If we aren't using animation, stick with single buffering, which is the default.

3. glutInitWindowSize(640,480)

We need to create the characteristics of our window. A call to glutInitWindowSize() will be used to specify the size, in pixels, of our initial window. The arguments indicate the height and width (in pixels) of the requested window.

4. glutInitWindowPosition(100,15)

Similarly, glutInitWindowPosition() is used to specify the screen location for the upper-left corner of our initial window. The arguments, x and y, indicate the location of the window relative to the entire display. This function positioned the screen 100 pixels over from the left edge and 150 pixels down from the top.

5. glutCreateWindow("Example")

To create a window, the with the previously set characteristics (display mode, size, location, etc), the programmer uses the glutCreateWindow() command. The command takes a string as a parameter which may appear in the title bar.

6. glutMainLoop()

The window is not actually displayed until the glutMainLoop() is entered. The very last thing is we have to call this function

Event Driven Programming

The method of associating a call back function with a particular type of event is called as event driven programming. OpenGL provides tools to assist with the event management.

There are four Glut functions available

1. glutDisplayFunc(mydisplay)

The glutDisplayFunc() procedure is the first and most important event callback function. A callback function is one where a programmer-specified routine can be registered to be called in response to a specific type of event. For example, the argument of glutDisplayFunc(mydisplay) is the function that is called whenever GLUT determines that the contents of the window needs to be redisplayed. Therefore, we should put all the routines that you need to draw a scene in this display callback function.

2. glutReshapeFunc(myreshape)

The `glutReshapeFunc()` is a callback function that specifies the function that is called whenever the window is resized or moved. Typically, the function that is called when needed by the reshape function displays the window to the new size and redefines the viewing characteristics as desired.

3. glutKeyboardFunc(mykeyboard)

GLUT interaction using keyboard inputs is handled. The command `glutKeyboardFunc()` is used to run the callback function specified and pass as parameters, the ASCII code of the pressed key, and the x and y coordinates of the mouse cursor at the time of the event.

Special keys can also be used as triggers. The key passed to the callback function, in this case, takes one of the following values (defined in `glut.h`).

Special keys can also be used as triggers. The key passed to the callback function, in this case, takes one of the following values (defined in `glut.h`).

- | | |
|----------------------|-------------|
| • GLUT_KEY_UP | Up Arrow |
| • GLUT_KEY_RIGHT | Right Arrow |
| • GLUT_KEY_DOWN | Down Arrow |
| • GLUT_KEY_PAGE_UP | Page Up |
| • GLUT_KEY_PAGE_DOWN | Page Down |
| • GLUT_KEY_HOME | Home |
| • GLUT_KEY_END | End |
| • GLUT_KEY_INSERT | Insert |

4. glutMouseFunc(mymouse)

GLUT supports interaction with the computer mouse that is triggered when one of the three typical buttons is pressed. A mouse callback function can be initiated when a given mouse button is pressed or released. The command `glutMouseFunc()` is used to specify the callback function to use when a specified button is in a given state at a certain location. These buttons are defined as either `GL_LEFT_BUTTON`, `GL_RIGHT_BUTTON`, or `GL_MIDDLE_BUTTON` and the states for that button are either `GLUT_DOWN` (when pressed) or `GLUT_UP` (when released). Finally, x and y callback parameters indicate the location (in window-relative coordinates) of the mouse at the time of the event.

Example : Skeleton for OpenGL Code

```
int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(465, 250);
    glutInitWindowPosition(100, 150);
    glutCreateWindow("My First Example");
    glutDisplayFunc(mydisplay);
    glutReshapeFunc(myreshape);
    glutMouseFunc(mymouse);
    glutKeyboardFunc(mykeyboard);
    myinit();
    glutMainLoop();
    return 0;
}
```

Basic graphics primitives

OpenGL Provides tools for drawing all the output primitives such as points, lines, triangles, polygons, quads etc and it is defined by one or more vertices.

To draw such objects in OpenGL we pass it a list of vertices. The list occurs between the two OpenGL function calls glBegin() and glEnd(). The argument of glBegin() determine which object is drawn.

These functions are

```
glBegin(int mode);
glEnd( void );
```

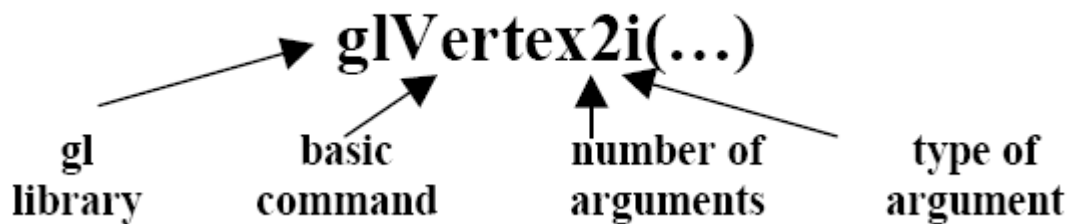
The parameter mode of the function glBegin can be one of the following:

```
GL_POINTS
GL_LINES
GL_LINE_STRIP
GL_LINE_LOOP
GL_TRIANGLES
GL_TRIANGLE_STRIP
GL_TRIANGLE_FAN
GL_QUADS
```

GL_QUAD_STRIP
GL_POLYGON

glVertex() : The main function used to draw objects is named as glVertex. This function defines a point (or a vertex) and it can vary from receiving 2 up to 4 coordinates.

Format of glVertex Command



When we wish to refer the basic command without regard to the specific arguments and datatypes it is specified as

```
glVertex*();
```

OpenGL Datatypes

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

Example

//the following code plots three dots

```
glBegin(GL_POINTS);  
glVertex2i(100, 50);  
glVertex2i(100, 130);  
glVertex2i(150, 130);  
glEnd( );
```

// the following code draws a triangle

```
glBegin(GL_TRIANGLES);  
glVertex3f(100.0f, 100.0f, 0.0f);  
glVertex3f(150.0f, 100.0f, 0.0f);  
glVertex3f(125.0f, 50.0f, 0.0f);  
glEnd( );
```

// the following code draw a lines

```
glBegin(GL_LINES);  
glVertex3f(100.0f, 100.0f, 0.0f); // origin of the line  
glVertex3f(200.0f, 140.0f, 5.0f); // ending point of the line  
glEnd( );
```

OpenGL State

OpenGL keeps track of many state variables, such as current size of a point, the current color of a drawing, the current background color, etc.

The value of a state variable remains active until new value is given.

glPointSize() : The size of a point can be set with glPointSize(), which takes one floating point argument

Example : glPointSize(4.0);

glClearColor() : establishes what color the window will be cleared to. The background color is set with glClearColor(red, green, blue, alpha), where alpha specifies a degree of transparency

Example : glClearColor (0.0, 0.0, 0.0, 0.0); //set black background color

glClear() : To clear the entire window to the background color, we use glClear (GL_COLOR_BUFFER_BIT). The argument GL_COLOR_BUFFER_BIT is another constant built into OpenGL

Example : glClear(GL_COLOR_BUFFER_BIT)

glColor3f() : establishes to use for drawing objects. All objects drawn after this point use this color, until it's changed with another call to set the color.

Example:

```
glColor3f(0.0, 0.0, 0.0);           //black
glColor3f(1.0, 0.0, 0.0);           //red
glColor3f(0.0, 1.0, 0.0);           //green
glColor3f(1.0, 1.0, 0.0);           //yellow
glColor3f(0.0, 0.0, 1.0);           //blue
glColor3f(1.0, 0.0, 1.0);           //magenta
glColor3f(0.0, 1.0, 1.0);           //cyan
glColor3f(1.0, 1.0, 1.0);           //white
```

gluOrtho2D(): specifies the coordinate system in two dimension

void gluOrtho2D (GLdouble *left*, GLdouble *right*, GLdouble *bottom*,GLdouble *top*);

Example : gluOrtho2D(0.0, 640.0, 0.0, 480.0);

glOrtho() : specifies the coordinate system in three dimension

Example : glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);

glFlush() : ensures that the drawing commands are actually executed rather than stored in a buffer awaiting (ie) Force all issued OpenGL commands to be executed

glMatrixMode(GL_PROJECTION) : For orthographic projection

glLoadIdentity() : To load identity matrix

glShadeModel : Sets the shading model. The mode parameter can be either GL_SMOOTH (the default) or GL_FLAT.

void glShadeModel (GLenum mode);

With flat shading, the color of one particular vertex of an independent primitive is duplicated across all the primitive's vertices to render that primitive. With smooth shading, the color at each vertex is treated individually.

Example : OpenGL Program to draw three dots (2-Dimension)

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void myInit(void)
{
    glClearColor (1.0, 1.0, 1.0, 0.0);
    glColor3f (0.0, 0.0, 0.0);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void Display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
    glVertex2i(100, 50);
    glVertex2i(100, 130);
    glVertex2i(150, 130);
    glEnd( );
    glFlush();
}
int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(100,150);
    glutCreateWindow("Example");
    glutDisplayFunc(Display);
    myInit();
    glutMainLoop();
    return 0;
}
```

Example : White Rectangle on a Black Background (3-Dimension co-ordinates)

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

void Display(void)
{
    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();
}

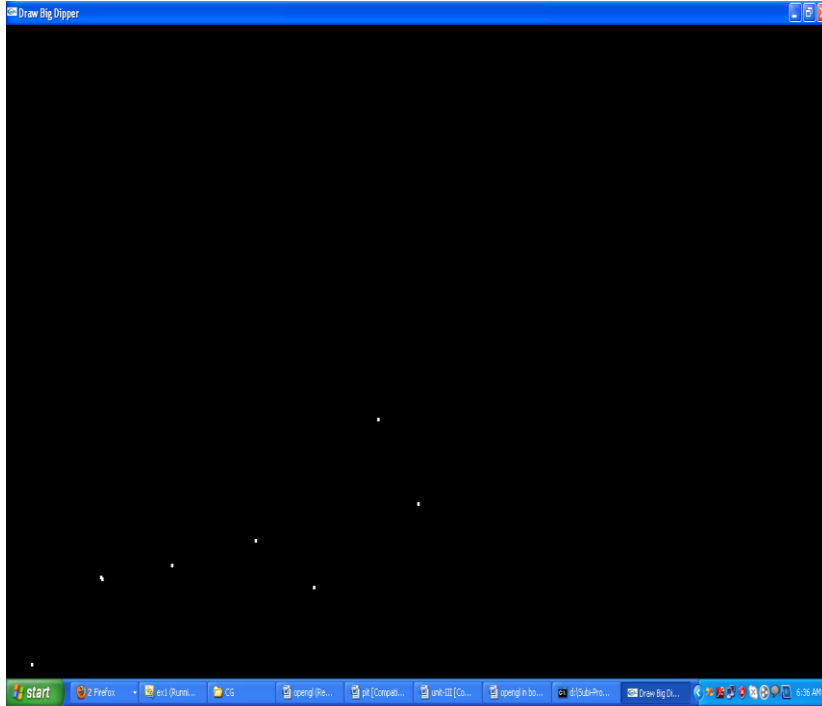
int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640,480);
    glutCreateWindow("Intro");
    glClearColor(0.0,0.0,0.0,0.0);
    glutDisplayFunc(Display);
    glutMainLoop();
    return 0;
}
```



Example : Big Dipper

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void myInit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glColor3f (1.0, 1.0, 1.0);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}
void Display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glBegin(GL_POINTS);
    glVertex2i(289, 190);
    glVertex2i(320, 128);
    glVertex2i(239, 67);
    glVertex2i(194, 101);
    glVertex2i(129, 83);
    glVertex2i(75, 73);
    glVertex2i(74, 74);
    glVertex2i(20, 10);
    glEnd( );
    glFlush();
}

int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(640,480);
    glutInitWindowPosition(100,150);
    glutCreateWindow("Draw Big Dipper");
    glutDisplayFunc(Display);
    myInit();
    glutMainLoop();
    return 0;
}
```



Making Line Drawings

OpenGL makes it easy to draw a line: use `GL_LINES` as the argument to `glBegin()`, and pass it the two end points as vertices. Thus to draw a line between (40,100) and (202,96) use:

```
glBegin(GL_LINES); // use constant GL_LINES here
glVertex2i(40, 100);
glVertex2i(202, 96);
glEnd();
```

OpenGL provides tools for setting the attributes of lines.

A line's color is set in the same way as for points, using `glColor3f()`.

To draw thicker lines use `glLineWidth(4.0)`. The default thickness is 1.0

To make stippled (dotted or dashed) lines, you use the command **`glLineStipple()`** to define the stipple pattern, and then we enable line stippling with **`glEnable()`**

```
glLineStipple(1, 0x3F07);
glEnable(GL_LINE_STIPPLE);
```

Drawing Polylines and Polygons

Polyline is a collection of line segments joined end to end. It is described by an ordered list of points,

$$p_0 = (x_0, y_0), p_1 = (x_1, y_1), \dots, p_n = (x_n, y_n).$$

In OpenGL a polyline is called a “line strip”, and is drawn by specifying the vertices in turn between `glBegin(GL_LINE_STRIP)` and `glEnd()`.

For example, the code:

```
glBegin(GL_LINE_STRIP); // draw an open polyline
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();
```

```
glBegin(GL_LINE_LOOP); // draw an polygon
glVertex2i(20,10);
glVertex2i(50,10);
glVertex2i(20,80);
glVertex2i(50,80);
glEnd();
glFlush();
```

Attributes such as color, thickness and stippling may be applied to polylines in the same way they are applied to single lines. If it is desired to connect the last point with the first point to make the polyline into a polygon simply replace `GL_LINE_STRIP` with `GL_LINE_LOOP`.

Polygons drawn using `GL_LINE_LOOP` cannot be filled with a color or pattern. To draw filled polygons we have to use `glBegin(GL_POLYGON)`

Drawing Aligned Rectangles.

A special case of a polygon is the **aligned rectangle**, so called because its sides are aligned with the coordinate axes.

OpenGL provides the ready-made function:

```
glRecti(GLint x1, GLint y1, GLint x2, GLint y2);

// draw a rectangle with opposite corners (x1, y1) and (x2, y2);
// fill it with the current color;

glClearColor(1.0,1.0,1.0,0.0); // white background
glClear(GL_COLOR_BUFFER_BIT); // clear the window
glColor3f(0.6,0.6,0.6); // bright gray
glRecti(20,20,100,70);
glColor3f(0.2,0.2,0.2); // dark gray
glRecti(70, 50, 150, 130);

aspect ratio = width/height;
```

Polygons

Polygons are the areas enclosed by single closed loops of line segments, where the line segments are specified by the vertices at their endpoints

Polygons are typically drawn by filling in all the pixels enclosed within the boundary, but you can also draw them as outlined polygons or simply as points at the vertices. A filled polygon might be solidly filled, or stippled with a certain pattern

OpenGL also supports filling more general polygons with a pattern or color.

To draw a convex polygon based on vertices (x_0, y_0) , (x_1, y_1) , ..., (x_n, y_n) use the usual list of vertices, but place them between a `glBegin(GL_POLYGON)` and an `glEnd()`:

```
glBegin(GL_POLYGON);
glVertex2f(x0, y0);
glVertex2f(x1, y1);
. . . . .
glVertex2f(xn, yn);
glEnd();
```

The following list explains the function of each of the five constants:

GL_TRIANGLES: takes the listed vertices three at a time, and draws a separate triangle for each;

GL_QUADS: takes the vertices four at a time and draws a separate quadrilateral for each

GL_TRIANGLE_STRIP: draws a series of triangles based on triplets of vertices: v_0, v_1, v_2 , then v_2, v_1, v_3 , then v_2, v_3, v_4 , etc. (in an order so that all triangles are “traversed” in the same way; e.g. counterclockwise).

GL_TRIANGLE_FAN: draws a series of connected triangles based on triplets of vertices: v_0, v_1, v_2 , then v_0, v_2, v_3 , then v_0, v_3, v_4 , etc.

GL_QUAD_STRIP: draws a series of quadrilaterals based on foursomes of vertices: first v_0, v_1, v_3, v_2 , then v_2, v_3, v_5, v_4 , then v_4, v_5, v_7, v_6 (in an order so that all quadrilaterals are “traversed” in the same way; e.g. counterclockwise).

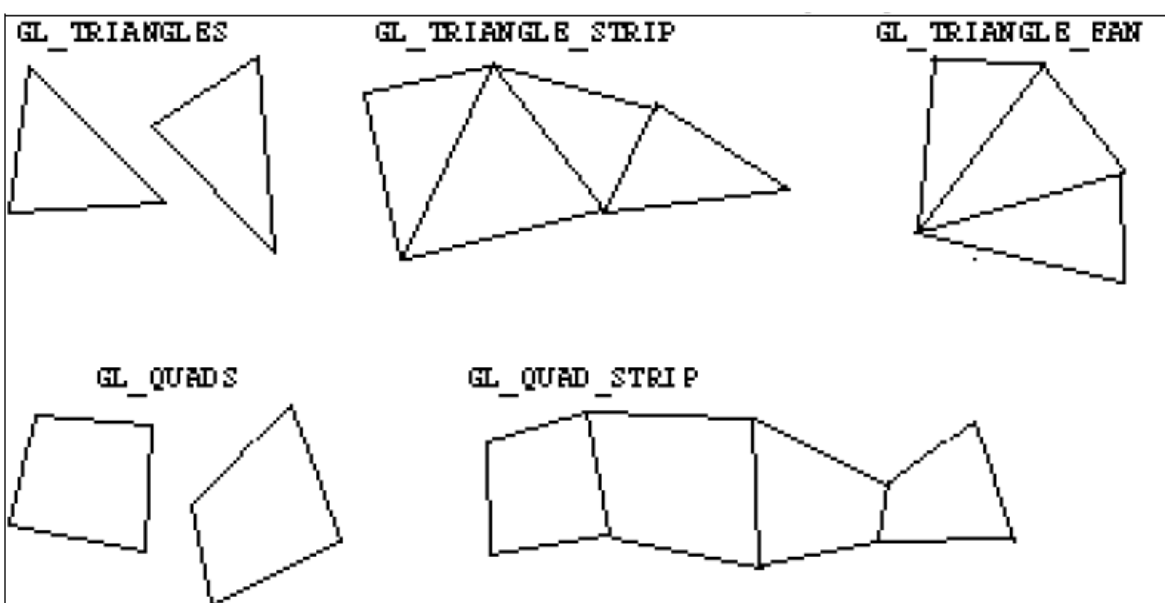


Figure 2.37. Other geometric primitive types.

Example to draw smooth shaded Triangle with shades

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
    gluOrtho2D (0.0, 640.0, 0.0, 480.0);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity ();
}
void display(void)
```



```

{
glClear (GL_COLOR_BUFFER_BIT);
glBegin (GL_TRIANGLES);
glColor3f (1.0, 0.0, 0.0);
glVertex2f (50.0, 50.0);
glColor3f (0.0, 1.0, 0.0);
glVertex2f (250.0, 50.0);
glColor3f (0.0, 0.0, 1.0);
glVertex2f (50.0, 250.0);
glEnd();
glFlush ();
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow ("Shade");
init ();
glutDisplayFunc(display);
glutMainLoop();
return 0;
}

```

Polygon Filling

A filled polygon might be **solidly filled**, or **stippled** with a certain pattern.

The pattern is specified with 128-byte array of data type GLubyte. The 128 bytes provides the bits for a mask that is 32 bits wide and 32 bits high.

GLubyte mask[]={0xff,0xfe.....128 entries}

The first 4 bytes prescribe the 32 bits across the bottom row from left to right; the next 4 bytes give the next row up, etc..

Example

```

#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>

```

```

    GLubyte mask[]={
        0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x03, 0x80, 0x01, 0xC0, 0x06, 0xC0, 0x03, 0x60,
        0x04, 0x60, 0x06, 0x20, 0x04, 0x30, 0x0C, 0x20,
        0x04, 0x18, 0x18, 0x20, 0x04, 0x0C, 0x30, 0x20,
        0x04, 0x06, 0x60, 0x20, 0x44, 0x03, 0xC0, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x44, 0x01, 0x80, 0x22, 0x44, 0x01, 0x80, 0x22,
        0x66, 0x01, 0x80, 0x66, 0x33, 0x01, 0x80, 0xCC,
        0x19, 0x81, 0x81, 0x98, 0x0C, 0xC1, 0x83, 0x30,
        0x07, 0xe1, 0x87, 0xe0, 0x03, 0x3f, 0xfc, 0xc0,
        0x03, 0x31, 0x8c, 0xc0, 0x03, 0x33, 0xcc, 0xc0,
        0x06, 0x64, 0x26, 0x60, 0x0c, 0xcc, 0x33, 0x30,
        0x18, 0xcc, 0x33, 0x18, 0x10, 0xc4, 0x23, 0x08,
        0x10, 0x63, 0xC6, 0x08, 0x10, 0x30, 0x0c, 0x08,
        0x10, 0x18, 0x18, 0x08, 0x10, 0x00, 0x00, 0x08};

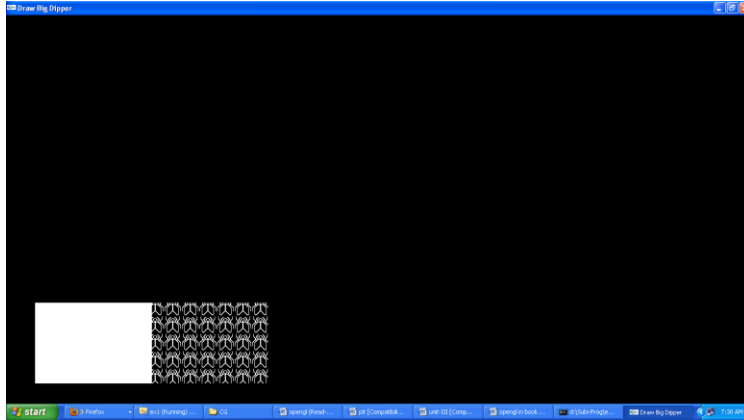
void myInit(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glColor3f (1.0, 1.0, 1.0);
    glPointSize(4.0);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluOrtho2D(0.0, 640.0, 0.0, 480.0);
}

void Display(void)
{
    glClearColor(0.0,0.0,0.0,0.0); // white background
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glRectf(25.0, 25.0, 125.0, 125.0);
    glEnable(GL_POLYGON_STIPPLE);
    glPolygonStipple(mask);
    glRectf (125.0, 25.0, 225.0, 125.0);
    glDisable(GL_POLYGON_STIPPLE);
    glFlush();
}

int main (int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

```

```
glutInitWindowSize(640,480);
glutInitWindowPosition(100,150);
glutCreateWindow("Polygon Stipple");
glutDisplayFunc(Display);
myInit();
glutMainLoop();
return 0;
}
```



Simple Interaction with the mouse and keyboard

When the user presses or releases a mouse button, moves the mouse, or presses a keyboard key, an event occurs. Using the OpenGL Utility Toolkit (GLUT) the programmer can register a callback function with each of these events by using the following commands:

glutMouseFunc(myMouse) which registers myMouse() with the event that occurs when the mouse button is pressed or released;

glutMotionFunc(myMovedMouse) which registers myMovedMouse() with the event that occurs when the mouse is moved while one of the buttons is pressed;

glutKeyboardFunc(myKeyboard) which registers myKeyBoard() with the event that occurs when a keyboard key is pressed.

Mouse interaction.

```
void myMouse(int button, int state, int x, int y);
```

When a mouse event occurs the system calls the registered function, supplying it with values for these parameters. The value of button will be one of:

GLUT_LEFT_BUTTON,
GLUT_MIDDLE_BUTTON,
GLUT_RIGHT_BUTTON,

with the obvious interpretation, and the value of state will be one of: GLUT_UP or GLUT_DOWN. The values x and y report the position of the mouse at the time of the event.

Keyboard interaction.

As mentioned earlier, pressing a key on the keyboard queues a keyboard event. The callback function myKeyboard() is registered with this type of event through

glutKeyboardFunc(myKeyboard).

It must have prototype:

void myKeyboard(unsigned int key, int x, int y);

The value of key is the ASCII value of the key pressed. The values x and y report the position of the mouse at the time that the event occurred. (As before y measures the number of pixels down from the top of the window.)

```
void myKeyboard(unsigned char theKey, int mouseX, int mouseY)
{
    GLint x = mouseX;
    GLint y = screenHeight - mouseY; // flip the y value as always
    switch(theKey)
    {
        case 'p':
            drawDot(x, y); // draw a dot at the mouse position
            break;
        case GLUT_KEY_LEFT: List[++last].x = x; // add a point
            List[last].y = y;
            break;
        case 'E':
            exit(-1); //terminate the program
        default:
            break; // do nothing
    }
}
```

Drawing three dimensional objects & Drawing three dimensional scenes

OpenGL has separate transformation matrices for different graphics features

glMatrixMode(GLenum mode), where mode is one of:

- **GL_MODELVIEW** - for manipulating model in scene
- **GL_PROJECTION** - perspective orientation
- **GL_TEXTURE** - texture map orientation

glLoadIdentity(): loads a 4-by-4 identity matrix into the current matrix

glPushMatrix() : push current matrix stack

glPopMatrix() : pop the current matrix stack

glMultMatrix () : multiply the current matrix with the specified matrix

glViewport() : set the viewport

Example : glViewport(0, 0, width, height);

gluPerspective() : function sets up a perspective projection matrix.

Format : gluPerspective(angle, asratio, ZMIN, ZMAX);

Example : gluPerspective(60.0, width/height, 0.1, 100.0);

gluLookAt() - view volume that is centered on a specified eyepoint

Example : gluLookAt(3.0, 2.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);

glutSwapBuffers () : glutSwapBuffers swaps the buffers of the current window if double buffered.

Example for drawing three dimension Objects

```
glBegin(GL_QUADS); // Start drawing a quad primitive
glVertex3f(-1.0f, -1.0f, 0.0f); // The bottom left corner
glVertex3f(-1.0f, 1.0f, 0.0f); // The top left corner
glVertex3f(1.0f, 1.0f, 0.0f); // The top right corner
glVertex3f(1.0f, -1.0f, 0.0f); // The bottom right corner
glEnd();
```

// Triangle

```
glBegin( GL_TRIANGLES );
glVertex3f( -0.5f, -0.5f, -10.0 );
glVertex3f( 0.5f, -0.5f, -10.0 );
glVertex3f( 0.0f, 0.5f, -10.0 );
glEnd();
```

// Quads in different colours

```
glBegin(GL_QUADS);
glColor3f(1,0,0); //red
glVertex3f(-0.5, -0.5, 0.0);
glColor3f(0,1,0); //green
glVertex3f(-0.5, 0.5, 0.0);
glColor3f(0,0,1); //blue
glVertex3f(0.5, 0.5, 0.0);
glColor3f(1,1,1); //white
glVertex3f(0.5, -0.5, 0.0);
glEnd();
```

GLUT includes several routines for drawing these three-dimensional objects:

- cone
- icosahedron
- teapot
- cube
- octahedron
- tetrahedron
- dodecahedron
- sphere
- torus

OpenGL Functions for drawing the 3D Objects

```
glutWireCube(double size);
glutSolidCube(double size);
glutWireSphere(double radius, int slices, int stacks);
glutSolidSphere(double radius, int slices, int stacks);
glutWireCone(double radius, double height, int slices, int stacks);
glutSolidCone(double radius, double height, int slices, int stacks);
glutWireTorus(double inner_radius, double outer_radius, int sides, int rings);
glutSolidTorus(double inner_radius, double outer_radius, int sides, int rings);
glutWireTeapot(double size);
glutSolidTeapot(double size);
```

3D Transformation in OpenGL

glTranslate () : multiply the current matrix by a translation matrix

```
glTranslated(GLdouble x, GLdouble y, GLdouble z);
void glTranslatef(GLfloat x, GLfloat y, GLfloat z);
```

x, y, z - Specify the x, y, and z coordinates of a translation vector.

If the matrix mode is either GL_MODELVIEW or GL_PROJECTION, all objects drawn after a call to glTranslate are translated.

Use glPushMatrix and glPopMatrix to save and restore the untranslated coordinate system.

glRotate() : multiply the current matrix by a rotation matrix

```
void glRotated(GLdouble angle, GLdouble x, GLdouble y, GLdouble z);
void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z);
```

angle : Specifies the angle of rotation, in degrees.

x, y, z : Specify the x, y, and z coordinates of a vector, respectively.

glScale() : multiply the current matrix by a general scaling matrix

```
void glScaled(GLdouble x, GLdouble y, GLdouble z);
void glScalef(GLfloat x, GLfloat y, GLfloat z);
```

x, y, z : Specify scale factors along the x, y, and z axes, respectively.

Example : Transformation of a Polygon

```
#include "stdafx.h"
#include "gl/glut.h"
#include <gl/gl.h>
void Display(void)
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glColor3f(0.0, 1.0, 0.0);
    glBegin(GL_POLYGON);
        glVertex3f( 0.0, 0.0, 0.0);    // V0 ( 0, 0, 0)
        glVertex3f( 1.0f, 0.0, 0.0);    // V1 ( 1, 0, 0)
```

```

        glVertex3f( 1.0f, 1.0f, 0.0);    // V2 ( 1, 1, 0)
        glVertex3f( 0.5f, 1.5f, 0.0);    // V3 (0.5, 1.5, 0)
        glVertex3f( 0.0, 1.0f, 0.0);    // V4 ( 0, 1, 0)
        glEnd();
        glPushMatrix();
        glTranslatef(1.5, 2.0, 0.0);
        glRotatef(90.0, 0.0, 0.0, 1.0);
        glScalef(0.5, 0.5, 0.5);
        glBegin(GL_POLYGON);
        glVertex3f( 0.0, 0.0, 0.0);    // V0 ( 0, 0, 0)
        glVertex3f( 1.0f, 0.0, 0.0);    // V1 ( 1, 0, 0)
        glVertex3f( 1.0f, 1.0f, 0.0);    // V2 ( 1, 1, 0)
        glVertex3f( 0.5f, 1.5f, 0.0);    // V3 (0.5, 1.5, 0)
        glVertex3f( 0.0, 1.0f, 0.0);    // V4 ( 0, 1, 0)
        glEnd();
        glPopMatrix();
    glFlush();
    glutSwapBuffers();
}
void Init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
}
void Resize(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.0, width/height, 0.1, 1000.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
int main(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(400, 400);
    glutInitWindowPosition(200, 200);
    glutCreateWindow("Polygon in OpenGL");
    Init();
    glutDisplayFunc(Display);
    glutReshapeFunc(Resize);
    glutMainLoop();
    return 0;
}

```