

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

3.1. COLOR MODELS

CONCEPTS:

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 spectrum.

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. 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 is the perceived intensity of the light.

Intensity is 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 is 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.

3.2.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

$C\lambda = X\mathbf{X} + Y\mathbf{Y} + Z\mathbf{Z}$ -----(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,

$$x = X/(X+Y+Z),$$

$$y = Y/(X+Y+Z),$$

$$z = Z/(X+Y+Z) \text{ with } x + y + z = 1$$

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.

RGB Color Model:

Based on the tristimulus theory of 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.

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

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.

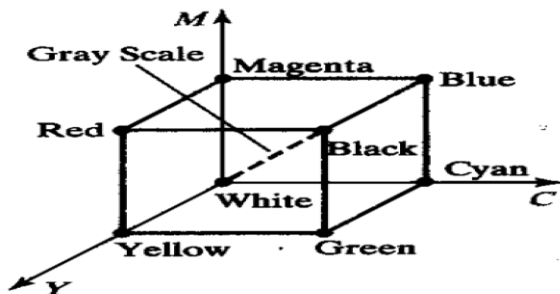
Parameter Q carries green-magenta hue information in a bandwidth of about 0.6 MHz. An RGB signal can be converted to a TV signal

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.

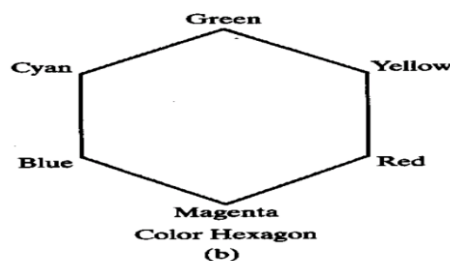
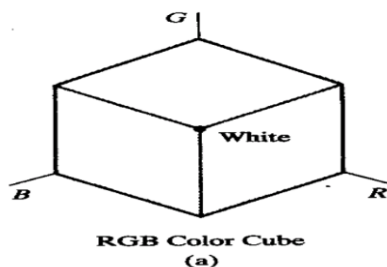
A combination of cyan and magenta ink produces blue light because the red and green components of the incident light are absorbed.

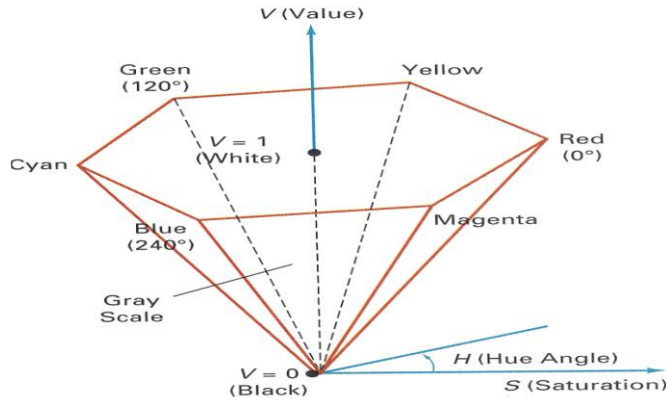
The printing process often used with the CMY model generates a color point with a collection of 4 ink dots; one dot is used for each of the primary colors (cyan, magenta and yellow) and one dot in black.

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.

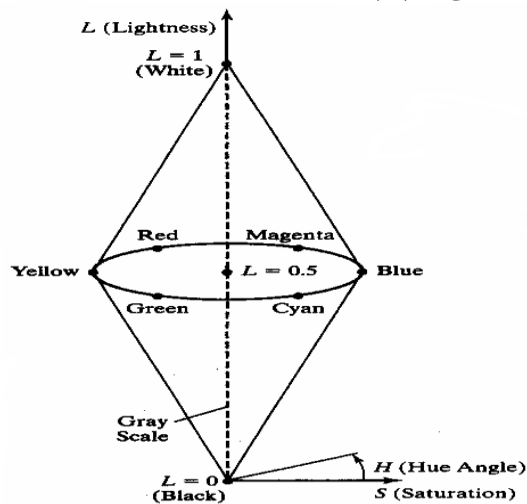




HLS Color Model

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).



3.3.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.

DESIGN OF ANIMATION SEQUENCES

In general, an animation sequence is designed with the following steps:

- Storyboard layout
- Object definitions

- Key-frame specifications
- Generation of in-between frames

This standard approach for animated cartoons is applied to other animation applications as well, although there are many special applications that do not follow this sequence. Real-time computer animations produced by flight simulators, for instance, display motion sequences in response to settings on the air-craft controls.

And visualization applications are generated by the solutions of the numerical models. For 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.

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

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. In addition, the associated movements for each object are specified along with the shape.

A keyframe is a 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 great. More key frames are specified for intricate motions than for simple, slowly varying motions.

In-betweens are the intermediate frames between the key frames. The number of in-betweens 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 second. Typically, time intervals for the motion are set up so that there are from three to five in-betweens for each pair of key frames. Depending on the speed specified for the motion, some key frames can be duplicated. For a 1-minute film sequence with no duplication, we would need 1440 frames. With five in-betweens for each pair of key frames, we would need 288 key frames. If the motion is not too complicated, we could space the key frames a little farther apart.

There are several other tasks that may be required, depending on the application.

They include motion verification, editing, and production and synchronization of a soundtrack. Many of the functions needed to produce general animations are now computer-generated. Figures 16-1 and 16-2 show examples of computer-generated frames for animation sequences

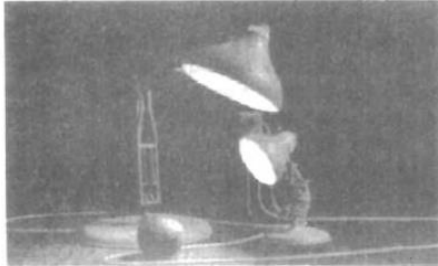


Figure 16-1
One frame from the award-winning computer-animated short film *Luxo Jr.* The film was designed using a key-frame animation system and cartoon animation techniques to provide lifelike actions of the lamps. Final images were rendered with multiple light sources and procedural texturing techniques. (Courtesy of Pixar. © 1986 Pixar.)



Figure 16-2
One frame from the short film *Tin Toy*, the first computer-animated film to win an Oscar. Designed using a key-frame animation system, the film also required extensive facial expression modeling. Final images were rendered using procedural shading, self-shadowing techniques, motion blur, and texture mapping. (Courtesy of Pixar. © 1988 Pixar.)

3.3.1 GENERAL COMPUTER-ANIMATION FUNCTIONS:

Some steps in the development of an animation sequence are well-suited to computer solution. These include object manipulations and rendering, camera motions, and the generation of in-betweens. Animation packages, such as Wavefront, for example, provide special functions for designing the animation and processing individual objects. One function available in animation packages is provided to store and manage the object database. Object shapes and associated parameters are stored and updated in the database. Other object functions include those for motion generation and those for object rendering. Motions can be generated according to specified constraints using two-dimensional or three-dimensional transformations.

Standard functions can then be applied to identify visible surfaces and apply the rendering algorithms.

Another typical function simulates camera movements. Standard motions are zooming, panning, and tilting. Finally, given the specification for the key frames, the in-between can be automatically generated

RASTER ANIMATIONS:

On raster systems, we can generate real-time animation in limited applications using raster operations. As we have seen in Section 5-8, a simple method for translation in the xy plane is to transfer a rectangular block of pixel values from one location to another. Twodimensional rotations in multiples of 90° are also simple to perform, although we can rotate rectangular blocks of pixels through arbitrary angles using antialiasing procedures. To rotate a block of pixels, we need to determine the percent of area coverage for those pixels that overlap the rotated block. Sequences of raster operations can be executed to produce real-time animation of either two-dimensional or three-dimensional objects, as long as we restrict the animation to motions in the projection plane. Then no viewing or visible-surface algorithms need be invoked.

We can also animate objects along two-dimensional motion paths using the color-table transformations. Here we predefine the object at successive positions along the motion path, and set the successive blocks of pixel values to color-table entries. We set the pixels at the first position of the object to "on" values, and we set the pixels at the other object positions to the background color. The animation is then 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.

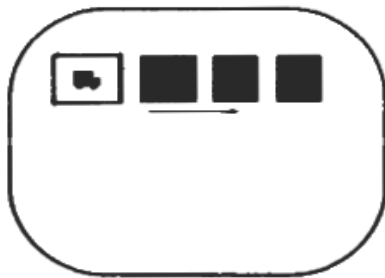


Figure 16-3
Real-time raster color-table
animation.

KEY FRAME SYSTEMS:

We generate each set of in-betweens from the specification of two (or more) key frames. Motionpaths can be given with a kinematic description as a set of spline curves, or the motions can be physically based by specifying the forces acting on the objects to be animated.

For complex scenes, we can separate the frames into individual components or objects called cels (celluloid transparencies), an acronym from cartoon animation. Given the animation paths, we can interpolate the positions of individual objects between any two times.

With complex object transformations, the shapes of objects may change over time. Examples are clothes, facial features, magnified detail, evolving shapes, exploding or disintegrating objects, and transforming one object into another object. If all surfaces are described with polygon meshes, then the number of edges per polygon can change from one frame to the next. Thus, the total number of line segments can be different in different frames.

Morphing:

Transformation of object shapes from one form to another is called morphing, which is a shortened form of metamorphosis. Morphing methods can be applied to any motion or transition involving a change in shape.

Given two key frames for an object transformation, we first adjust the object specification in one of the frames so that the number of polygon edges (or the number of vertices) is the same for the two frames. This preprocessing step is illustrated in Fig. 16-6. A straight-line segment in key frame k is transformed into two line segments in key frame $k+1$. Since key frame $k + 1$ has an extra vertex, we add a vertex between 1 and 2 in key frame k to balance the number of vertices (and edges) in the two key frames. Using linear interpolation to generate the in-betweens, we transition the added vertex in key frame k into vertex 3' along the straight-line path shown in Fig. 16-7. An example of a triangle linearly expanding into a quadrilateral is given in Fig. 16-8. Figures 16-9 and 16-10 show examples of morphing in television advertising.

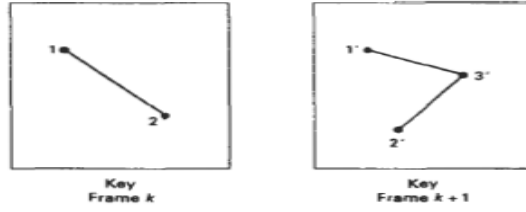


Figure 16-6
An edge with vertex positions 1 and 2 in key frame k evolves into two connected edges in key frame $k + 1$.

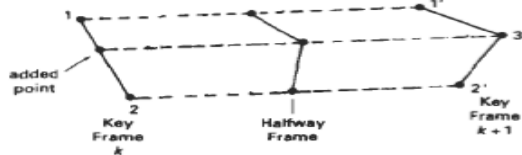


Figure 16-7
Linear interpolation for transforming a line segment in key frame k into two connected line segments in key frame $k + 1$.

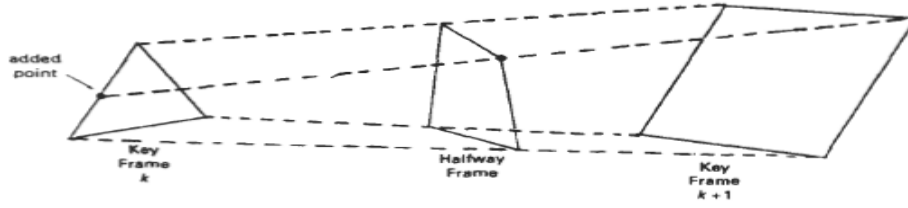


Figure 16-8
Linear interpolation for transforming a triangle into a quadrilateral.

We can state general preprocessing rules for equalizing key frames in terms of either the number of edges or the number of vertices to be added to a key frame. 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 then define

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

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

$$N_s = \text{int}\left(\frac{L_{\max}}{L_{\min}}\right)$$

Then the preprocessing is accomplished by

1. dividing N_e edges of $\text{keyframe}_{\min}$ into $N_s + 1$ sections
2. dividing the remaining lines of $\text{keyframe}_{\min}$ into N_s sections

As an example, if $L_k = 15$ and $L_{k+1} = 11$, we would divide 4 lines of keyframe_{k+1} into 2 sections each. The remaining lines of keyframe_{k+1} are left intact.

If we equalize the vertex count, we can use parameters V_k and V_{k+1} to denote the number of vertices in the two consecutive frames. In this case, we define

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

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

$$N_p = \text{int}\left(\frac{V_{\max} - 1}{V_{\min} - 1}\right)$$

Preprocessing using vertex count is performed by

1. adding N_p points to N_{ls} line sections of keyframe,,
2. adding $N_p - 1$ points to the remaining edges of keyframe,,

For the triangle-toquadrilateral example, $V_k = 3$ and $V_{k+1} = 4$. Both N_{ls} and N_p are 1, so we would add one point to one edge of keyframe $_k$. No points would be added to the remaining lines of keyframe $_{k+1}$.

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 16-11 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 (Fig. 16-12). The time interval between key frames is then divided into $n + 1$ subintervals, yielding an in-between spacing of

$$\Delta t = \frac{t_2 - t_1}{n + 1}$$

We can calculate the time for any in-between as

$$tB_j = t_1 + j \Delta t, \quad j = 1, 2, \dots, n$$

and determine the values for coordinate positions, color, and other physical parameters.

Nonzero accelerations are used to produce realistic displays of speed changes, particularly at the beginning and end of a motion sequence. We can model the start-up and slowdown portions of an animation path with spline or

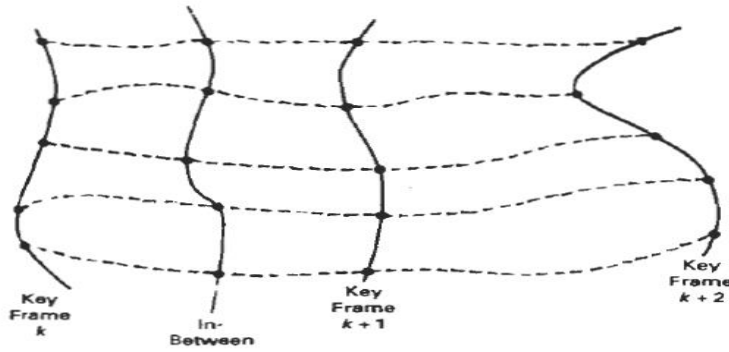


Figure 16-11
Fitting key-frame vertex positions with nonlinear splines.

trigonometric functions. Parabolic and cubic time functions have been applied to acceleration modeling, but trigonometric functions are more commonly used in animation packages.

To model increasing speed (positive acceleration), we want the time spacing between frames to increase so that greater changes in position occur as the object moves faster. We can obtain an increasing interval size with the

function $1 - \cos\theta$, $0 < \theta < \pi/2$

For n in-betweens, the time for the j th in-between would then be calculated as

$$tB_j = t_1 + \Delta t \left[1 - \cos \frac{j\pi}{2(n+1)} \right], \quad j = 1, 2, \dots, n$$

where Δt is the time difference between the two key frames. Figure 16-13 gives a plot of the trigonometric acceleration function and the in-between spacing for $n = 5$.

We can model decreasing speed (deceleration) with sine in the range $0 < \theta < \pi/2$. The time position of an in-between is now defined as

$$tB_j = t_1 + \Delta t \sin \frac{j\pi}{2(n+1)}, \quad j = 1, 2, \dots, n$$

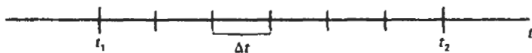


Figure 16-12
In-between positions for motion at constant speed.

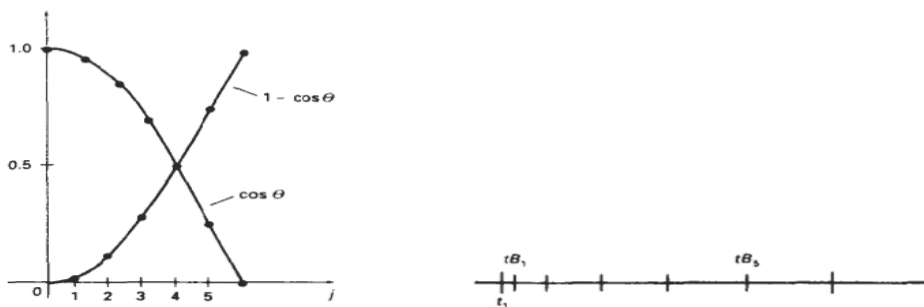


Figure 16-13
A trigonometric acceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-7, producing increased coordinate changes as the object moves through each time interval.

A plot of this function and the decreasing size of the time intervals is shown in Fig. 16-14 for five in-betweens.

Often, motions contain both speed-ups and slow-downs. We can model a combination of increasing-decreasing speed by first increasing the in-between time spacing, then we decrease this spacing. A function to accomplish these time changes is

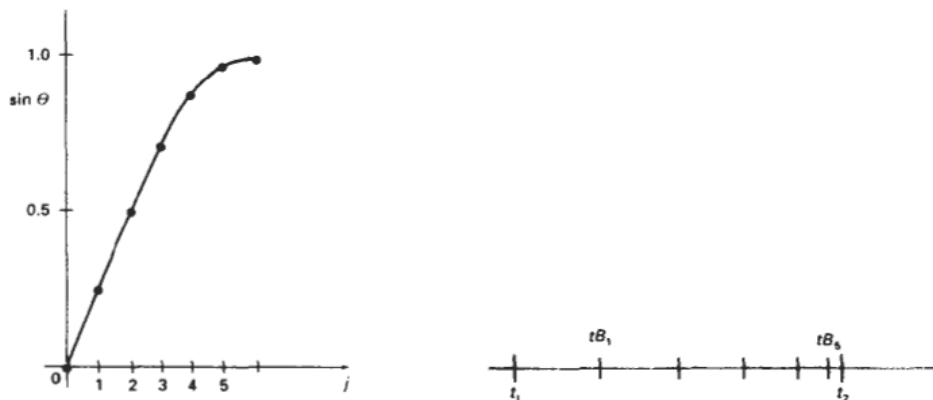


Figure 16-14
A trigonometric deceleration function and the corresponding in-between spacing for $n = 5$ and $\theta = j\pi/12$ in Eq. 16-8, producing decreased coordinate changes as the object moves through each time interval.

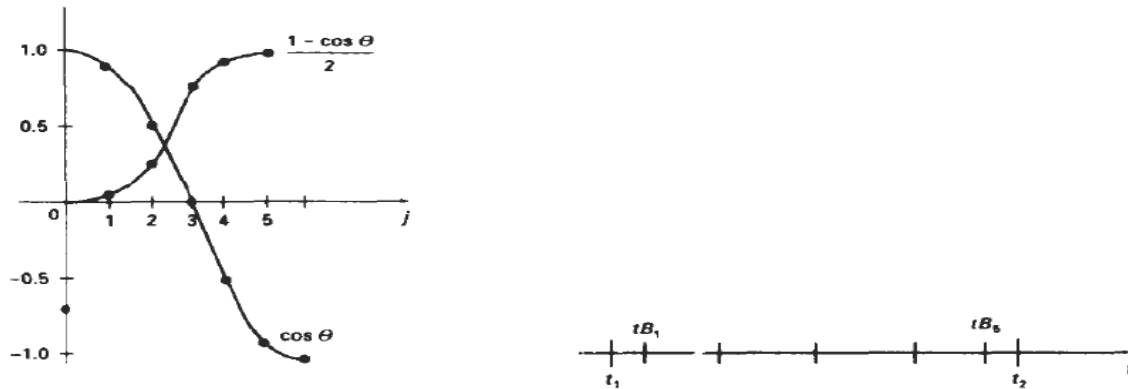


Figure 16-15

A trigonometric accelerate-decelerate function and the corresponding in-between spacing for $n = 5$ in Eq. 16-9.

$$\frac{1}{2}(1 - \cos \theta), \quad 0 < \theta < \pi/2$$

The time for the j th in-between is now calculated as

$$tB_j = t_1 + \Delta t \left\{ \frac{1 - \cos[j\pi/(n+1)]}{2} \right\}, \quad j = 1, 2, \dots, n \quad (16-9)$$

with Δt denoting the time difference for the two key frames. Time intervals for the moving object first increase then the time intervals decrease, as shown in Fig. 16-15.

Processing the in-betweens is simplified by initially modeling "skeleton" (wireframe) objects. This allows interactive adjustment of motion sequences. After the animation sequence is completely defined, objects can be fully rendered

3.5. GRAPHICS PROGRAMMING USING OpenGL

CONCEPT:

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).

Library 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

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.

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

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); //
gray glColor3f(0.2,0.2,0.2);
```

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.

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;

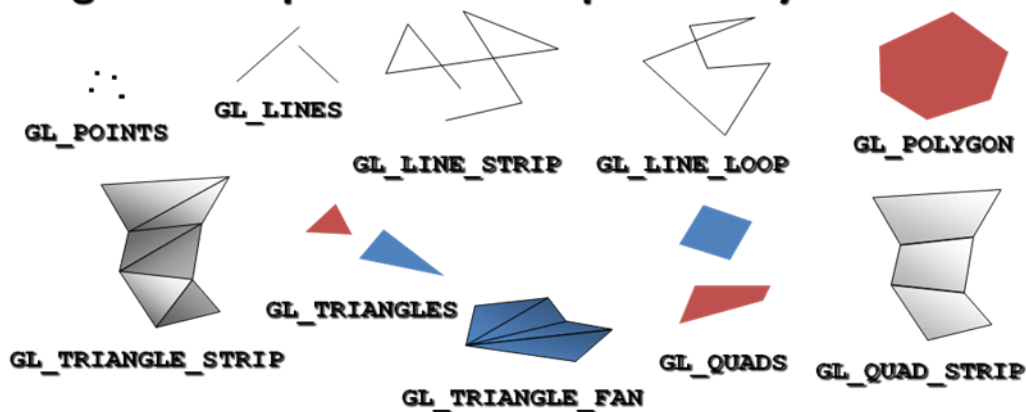
GL_QUADS: takes the vertices four at a time and draws a separate quadrilateral for each

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).

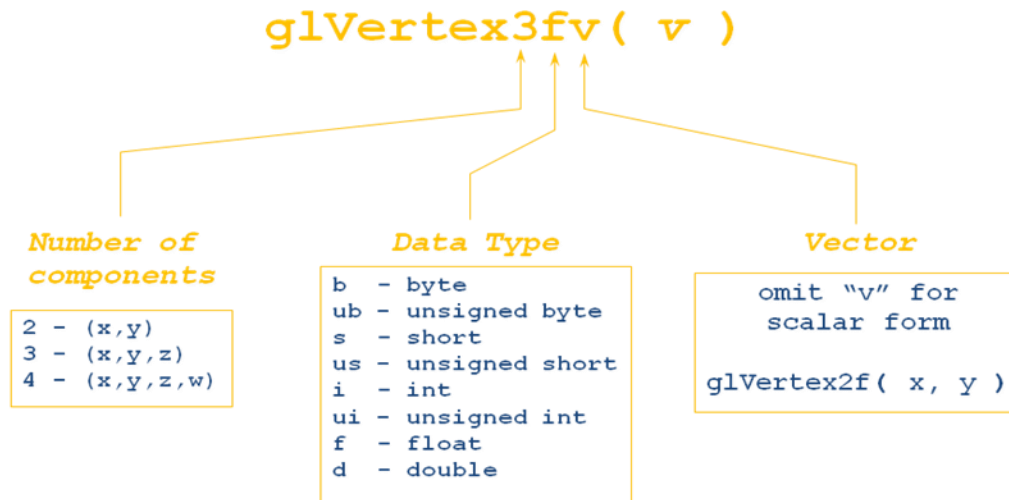
OpenGL Geometric Primitives

All geometric primitives are specified by vertices



21

OpenGL Command Formats:



Working With Material Properties In OpenGL

The effect of a light source can be seen only when light reflects off an object's surface. OpenGL provides methods for specifying the various reflection coefficients. The coefficients are set with variations of the function `glMaterial` and they can be specified individually for front and back faces.

The code: `GGLfloat myDiffuse[]={0.8, 0.2, 0.0, 1.0};`

`glMaterialfv(GL_FRONT, GL_DIFFUSE, myDiffuse);`

sets the diffuse reflection coefficients (p_{dr}, p_{dg}, p_{db}) equal to (0.8, 0.2, 0.0) for all specified front faces.

The first parameter of `glMaterialfv()` can take the following values: `GL_FRONT`: Set the reflection coefficient for front faces. `GL_BACK`: Set the reflection coefficient for back faces.

`GL_FRONT_AND_BACK`: Set the reflection coefficient for both front and back faces. The second parameter can take the following values:

`GL_AMBIENT`: Set the ambient reflection coefficients. `GL_DIFFUSE`: Set the diffuse reflection coefficients. `GL_SPECULAR`: Set the specular reflection coefficients.

`GL_AMBIENT_AND_DIFFUSE`:

Set both the ambient and the diffuse reflection coefficients to the same values.

`GL_EMISSION`: Set the emissive color of the surface. The emissive color of a face causes it to "glow" in the specified color, independently of any light source.

3.6 .DRAWING THREE DIMENSIONAL OBJECTS & DRAWING THREE DIMENSIONAL SCENES

CONCEPTS:

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**

// **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();

APPLICATIONS:

- 1.Implement a color models.
- 2.Implement a realistic scenes and objects
- 3.OPENGGL is a easy way to make a real objects.