Q.2 a. Define refresh buffer or frame buffer. Also define Aspect Ratio and Parametric continuity. (8)

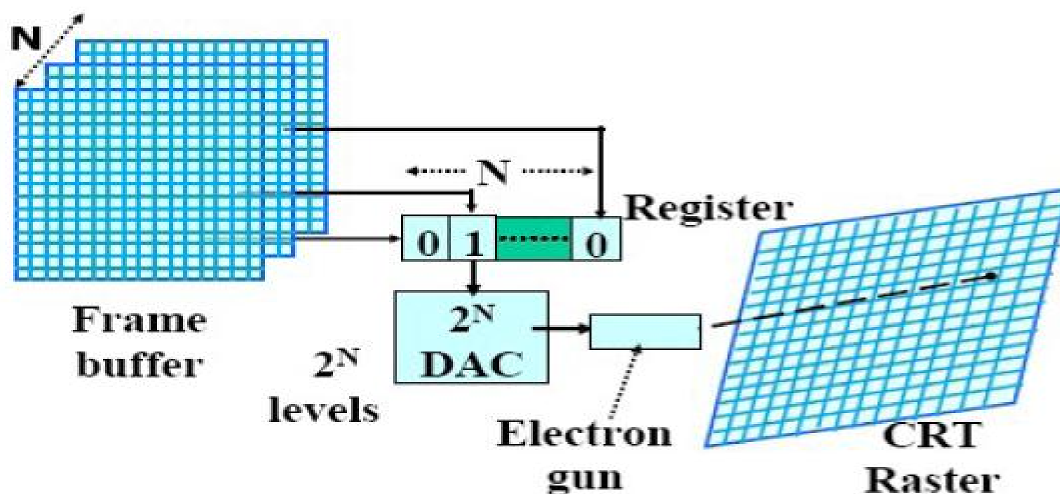
Answer:

A frame buffer is a large, contiguous piece of computer memory. At a minimum there is one memory bit for each pixel in the rater; this amount of memory is called a bit plane. The picture is built up in the frame buffer one bit at a time. You know that a memory bit has only two states, therefore a single bit plane yields a black-and white display. You know that a frame buffer is a digital device and the CRT is an analog device. Therefore, a conversion from a digital representation to an analog signal must take place when information is read from the frame buffer and displayed on the raster CRT graphics device. For this you can use a digital to analog converter (DAC). Each pixel in the frame buffer must be accessed and converted before it is visible on the raster CRT.

N-bit colour Frame buffer

Color or gray scales are incorporated into a frame buffer rater graphics device by using additional bit planes. The intensity of each pixel on the CRT is controlled by a corresponding pixel location in each of the N bit planes. The binary value from each of the N bit planes is loaded into corresponding positions in a register. The resulting binary number is interpreted as an intensity level between 0 (dark) and 2^n -1 (full intensity).

This is converted into an analog voltage between 0 and the maximum voltage of the electron gun by the DAC. A total of 2^N intensity levels are possible. Figure given below illustrates a system with 3 bit planes for a total of 8 (2^3) intensity levels. Each bit plane requires the full complement of memory for a given raster resolution; e.g., a 3-bit plane frame buffer for a 1024 X1024 raster requires 3,145,728 (3 X 1024 X1024) memory bits.



An N- bit plane gray level frame buffer

An increase in the number of available intensity levels is achieved for a modest increase in required memory by using a lookup table. Upon reading the bit planes in the frame buffer, the

resulting number is used as an index into the lookup table. The look up table must contain 2^N entries. Each entry in the lookup table is W bit wise. W may be greater than N. When this occurs, 2^W intensities are available; but only 2^N different intensities are available at one time. To get additional intensities, the lookup table must be changed.

Aspect ratio is a fancy term for "proportion," or the ratio of width to height. for example 4:3 for a computer screen. For instance, if a direction in a software manual tells you to "hold down the Shift key while you resize a graphic in order to maintain the aspect ratio," it simply means that if you don't hold down the Shift key you will stretch the image out of proportion.

Some combinations of computers and printers have trouble maintaining the correct aspect ratio when the image goes from the screen to the printer, or when the image is transferred from one system to another, so the aspect ratio can be an important specification to consider when choosing hardware.

The aspect ratio of the screen determines the most efficient screen RESOLUTIONS and the most desirable shape for individual PIXELS, all of which may have to change upon the introduction of HIGH DEFINITION TELEVISION.

Image resolution is the detail an image holds. The term applies to raster digital images, film images, and other types of images. Higher resolution means more image detail.

Image resolution can be measured in various ways. Basically, resolution quantifies how close lines can be to each other and still be visibly resolved. Resolution units can be tied to physical sizes (e.g. lines per mm, lines per inch), to the overall size of a picture (lines per picture height, also known simply as lines, TV lines, or TVL), or to angular subtenant. Line pairs are often used instead of lines; a line pair comprises a dark line and an adjacent light line. A line is either a dark line or a light line. A resolution 10 lines per millimeter means 5 dark lines alternating with 5 light lines, or 5 line pairs per millimeter (5 LP/mm). Photographic lens and film resolution are most often quoted in line pairs per millimeter.

b. Describe the working methodology of various input devices used for developing graphics applications. (8)

Answer: Input Devices

Input devices generate commands to control a process, such as the definition of a picture (indirectly by modifying a database). Input devices are logical devices. Their physical implementation might take various forms. They have nothing to do with output devices, though they might share some hardware or a communication link.

The command inlet to a process (either interactive or not) is an input stream (a file). Commands are structured or not, and are accompanied by data in various forms: text, numbers, arrays, etc. Commands are generated on a lower level by input tools.

Input tools - again an abstraction can be classified according to the type of data they deliver:-

Text tools (keyboard, voice)
Logical tools {function key)
1-D tools (control dials)
2-D tools (tracking cross, tablet)

3-D tools (3D joystick, Lincoln wand)
Name stack (lightpen, correlation)
Time
There has been an unfortunate preoccupation with the light pen as an input device and this may account for the relatively slow development of input devices compared with displays. The mouse or tracking ball or other related potentiometer-activating devices have found uses related to specific displays. The tablet in its various forms is however giving the user the natural freedom experienced with pencil and paper. A tablet is ideal for drawing-type self expression but is not the complete answer for all forms of interaction. The touch wire device originally developed for rapid interaction in air traffic control systems has an obvious place in normal input work for interacting with display menus. An extension to this is a proximity switch keyboard for the input of alpha-numeric data. Thus input devices attached to a display will be matched to specific functions rather than attempt to produce an all embracing single device. Having established a more than possible base for growth there are two major input areas that require more elegant solutions than are available at present. These are the rapid input of mass drawing data and the input of data related to three dimensional shapes, it is speculated that the work of J. Radon that resulted in tomography could be developed such that an object is placed upon a turn table, multiple projections taken and converted such that a complete three dimensional representation is loaded to computer store. The combined use of light and X-rays could provide for the collection of internal as well as external object data.

If a representation of an object is to be obtained from an engineering drawing drawn in an orthographic projection the problems are not trivial. They will be solved and drawing scanners having this capability will become common place.

Because of the volume of data and speed of processing required, these types of input units will have their own computing systems based upon microcomputers.

To summarise the ideas concerning input

The use of raster scan or related systems in reconstruction with picture processing, scene analysis and 3-D reconstruction will become one of the most important input devices to graphic systems in the future.

Techniques will be developed for computer recognition and refining of sloppy and incomplete drawings and models without the user having to be more explicit or categorical than he would be in communication with one of his colleagues.

Simple, user-friendly, adaptive command languages will be developed. They will make it possible to adapt the guidance, the commands and the error handling to the individual user's experience, skill and habits. Analogous to normal computer peripherals, graphical devices will become as self-contained as possible. They will have their own (more advanced) picture compiler. This will probably be a microprocessor, preferably programmable (local or remote) to accept picture descriptions of different standards.

Looking at these aspects from the viewpoint of a general user, there will be required a vast effort to produce graphics systems in which all of this related processing is transparent.

© IETE

Starting with pen and paper moving relative to each other under computer control one has the digital incremental plotter in all its forms and accuracies. These units are becoming faster in operation and the inherent mechanical problems of control and overshoot, acceleration against inertias and inkflow to the pen are increasing. The pen has in some devices been replaced by light or electron beams and the paper by some photosensitive material or the drum of a XEROX type copier.

At first sight the natural extensions appear to be in the direction of bigger and faster and perhaps cheaper. With the increase in the use of raster scan C.R.T. displays there will be an increasing need for hard copy units attached to such devices. It is thus envisaged that the XEROX type copier will be developed for this purpose in order to produce large high quality colour prints.

If this development trend results, what will become of ink jet type displays? A future is seen for these devices in the area of commercial art. Thus sizes will be increased. Research into the chemistry of the inks could well result in the ability to produce textured surfaces and the possibility of creating instant old masters! These types of display units could also become mass production devices by becoming substitutes for litho techniques for producing relatively small quantities of high quality output picture material.

An extension to these concepts could be the display of pictures by eroding multilayered material. Consider the use of a simple type of scraper board having a black top surface, white secondary surface followed by red, blue and green surfaces. A particular coloured line would be produced dependant upon the depth of a scribing tool. Much effort has been devoted to producing the illusion of three dimensional form using two dimensional displays. Perhaps the culmination of this effort is the work at the University of Utah, where the viewer who wears a special head-set, is presented at each eye with an image from a small C.R.T. By detecting the viewer's position the images are updated and the viewer can stroll around a virtual image.

Such a tool is ideal for research purposes and could play a vital role in psycho-analysis in that effects upon the mind could be created without recourse to narcotics. It is however not the sort of tool that would be used by a company board meeting discussing a new style of automobile.

One solution to this problem is to use a simple on-line machine tool and cut forms in plastic foam, chalk or other suitable media. Such a device is a hard copy unit and does not provide interaction, but it does provide a starting point for speculation. Surface production for a specific class of surfaces could be obtained by having a matrix of say, 1m × 1m which consists of rods of 0.5 mm diameter at 1 mm centres, with the possibility that the whole could be covered by an elastic membrane. Thus by pushing up the rods, representation of a surface can be obtained. This device could also be used in the dynamic sense to simulate vibration of surfaces and three dimensional wave motion. Another possibility for three dimensional display is the use of an electro-chromatic gel in the form of a block with electrodes attached on two sides. Surfaces would appear as surfaces of colour within the block. Interaction could be by a thin wand inserted into the block. Withdrawal of the wand causing the gel to close up. Such a concept is within the realms of possibility in that the Kerr cell uses electro-chromatic effects to produce a shutter for ultra-high speed cameras.

Another exciting possibility is the development of computer generated holograms. If such images are to be produced in virtually real time, new techniques of processor design will be

© IETE

required and parallel processing through the use of banks of microprocessors will probably be employed.

The ability to incorporate movement and interaction into computer displays came with the first refresh displays, and in essence these displays have altered little in their fundamentals.

Features that were implemented on a software basis are now incorporated as hardware and the introduction of microprocessors will ensure that such moves will continue as part of the development process. Colour is achieved with these displays using either phosphor layers of different colours which are excited by electron beams of varying intensity or shadow masks as in a colour television receiver. In order to give displayed output body there is a natural move toward raster scan systems and it is envisaged that the addition of video recording to such displays will be a natural development. The coupling of these two devices would provide a more flexible system for producing animation.

The rapid playback and editing facilities accorded by video recording of colour raster scan displays should produce a large impact on the area of film making. Another possibility would be the development of faster CRT that make it possible to draw 50-100,000 vectors at a 60-HZ refresh rate. This would give the possibility of drawing and manipulating reasonable complex pictures or making real-time computer animation.

High speed video recording and play back to other displays could be also used to provide a different level of multiplexing on systems having many users.

The related developments with Computer Output on Microfilm will be dependent upon developments in film technology with the present speed of processing colour film being a holding factor.

New applications of the physical properties of matter could be used in new displays. Examples of this could be large area liquid crystal displays and displays constructed from a matrix of micro light emitting diodes. Finally one should not preclude the possibility of direct interaction with the human brain.

To summarise the ideas concerning picture display.

Display devices will probably never be really cheap. One always wants the picture to be better, or to be produced faster.

A picture can be produced by:

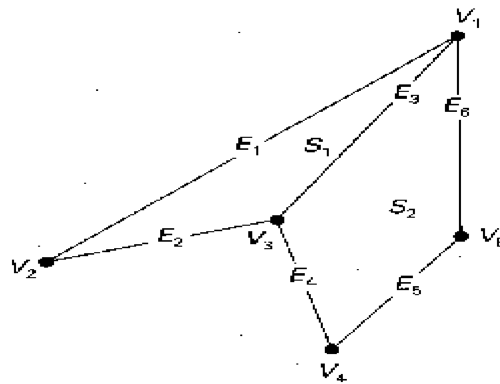
projection of visible light rays through an optical system into the retina
direct stimulation of brain cells (not necessarily through electrodes).
The latter possibility seems to have enormous potential, but will probably not be feasible in the near future. Existing devices are all of the first category. They either create a 2D or 3D object that is illuminated by ambient light (plotter, NC-machine), or they create the light rays themselves (CRT-screen, microfilm, hologram).

© IETE

Q.3 a. How do we represent polygon using polygon table, edge table and vertex table? Explain with example. (8)

Answer: Polygon Tables

- We specify objects as a set of vertices and associated attributes. This information can be stored in tables, of which there are two types: geometric tables and attribute tables.
- The geometry can be stored as three tables: a vertex table, an edge table, and a polygon table. Each entry in the vertex table is a list of coordinates defining that point. Each entry in the edge table consists of a pointer to each endpoint of that edge. And the entries in the polygon table define a polygon by providing pointers to the edges that make up the polygon.



VERTEX TABLE	
V_1:	x_1, y_1, z_1
V_2:	x_2, y_2, z_2
V_3:	x_3, y_3, z_3
V_4:	x_4, y_4, z_4
V_5:	x_5, y_5, z_6

EDGE TABLE	
E_1:	V_1, V_2
E_2:	V_2, V_3
E_3:	V_3, V_1
E_4:	V_3, V_4
E_5:	V_4, V_5
E_6:	V_5, V_1

POLYGON-SURFACE TABLE	
S_1:	E_1, E_2, E_3
S_2:	E_3, E_4, E_5, E_6

- We can eliminate the edge table by letting the polygon table reference the vertices directly, but we can run into problems, such as drawing some edges twice, because we don't realize that we have visited the same set of points before, in a different polygon. We could go even further and eliminate the vertex table by listing all the coordinates explicitly in the polygon table, but this wastes space because the same points appear in the polygon table several times.
- Using all three tables also allows for certain kinds of error checking. We can confirm that each polygon is closed, that each point in the vertex table is used in the edge table and that each edge is used in the polygon table.
- Tables also allow us to store additional information. Each entry in the edge table could have a pointer back to the polygons that make use of it. This would allow for quick look-up of those edges which are shared between polygons. We could also store the slope of each edge or the bounding box for each polygon--values which are repeatedly used in rendering and so would be handy to have stored with the data.

Example: Plane Equations

- Often in the graphics pipeline, we need to know the orientation of an object. It would be useful to store the plane equation with the polygons so that this information doesn't have to be computed each time.
- The plane equation takes the form:

  $Ax + By + Cz + D = 0$

  Using any three points from a polygon, we can solve for the coefficients. Then we can use the equation to determine whether a point is on the inside or outside of the plane formed by this polygon:

  $Ax + By + Cz + D < 0$ ==> inside
  $Ax + By + Cz + D > 0$ ==> outside

- The coefficients A, B, and C can also be used to determine a vector normal to the plane of the polygon. This vector, called the surface normal, is given simply by:

  $N = (A, B, C)$.

- If we specify the vertices of a polygon counterclockwise when viewing the outer side, in a right-handed coordinate system, the surface normal N will point from inside to outside. You can verify this from an alternate definition for N, based on three vertices:

  $N = (V2 - V1) \times (V3 - V1) = (A, B, C)$

  If we find N in this way, we still need D to complete the plane equation. The value of D is simply the dot product of the surface normal with any point in the polygon:

  $N . P = -D$

b. Explain the pipeline for transforming a view of a world-coordinate scene to device coordinates. Discuss the three-dimensional composite transformation **(8)**

Answer:

The purpose of the graphics pipeline is to create images and display them on your screen. The graphics pipeline takes geometric data representing an object or scene (typically in three dimensions) and creates a two-dimensional image from it. Your application supplies the geometric data as a collection of vertices that form polygons, lines, and points. The resulting image typically represents what an observer or camera would see from a particular vantage point.

As the geometric data flows through the pipeline, the GPU's vertex processor transforms the constituent vertices into one or more different coordinate systems, each of which serves a particular purpose. Cg vertex programs provide a way for you to program these transformations yourself.

Vertex programs may perform other tasks, such as lighting (discussed in Chapter 5) and animation (discussed in Chapter 6), but transforming vertex positions is a task*required* by all vertex programs. You cannot write a vertex program that does not output a transformed position,

because the rasterizer needs transformed positions in order to assemble primitives and generate fragments.

So far, the vertex program examples you've encountered limited their position processing to simple 2D transformations. This chapter explains how to implement conventional 3D transformations to render 3D objects.

Figure 4-1 illustrates the conventional arrangement of transforms used to process vertex positions. The diagram annotates the transitions between each transform with the coordinate space used for vertex positions as the positions pass from one transform to the next.

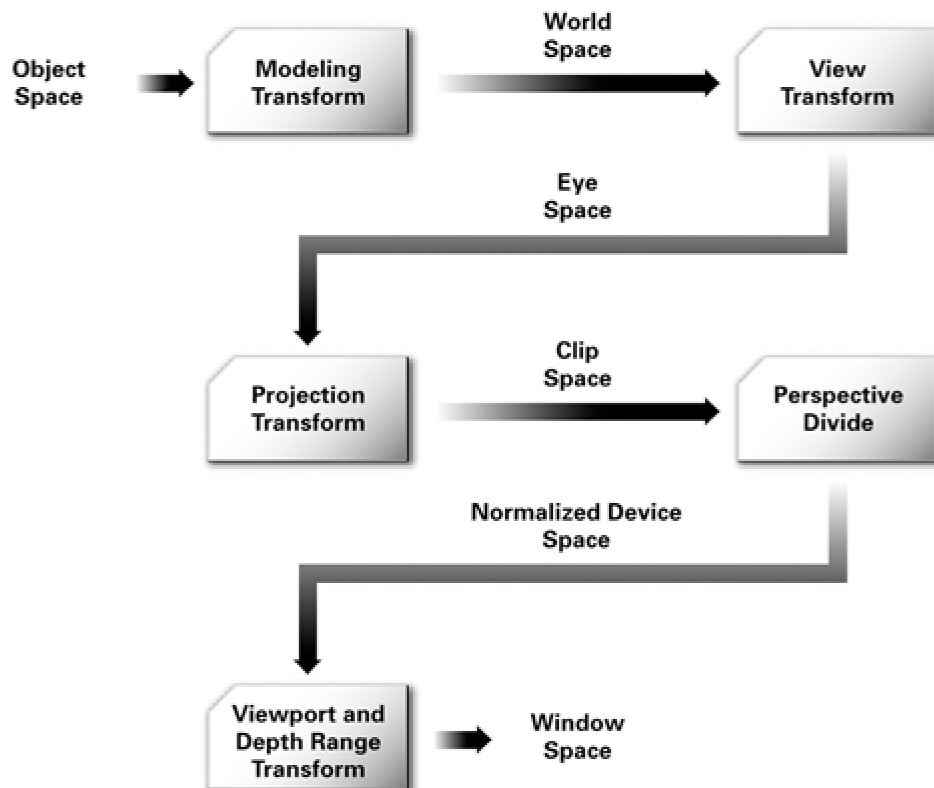


Figure 4-1 Coordinate Systems and Transforms for Vertex Processing

The following sections describe each coordinate system and transform in this sequence. We assume that you have some basic knowledge of matrices and transformations, and so we explain each stage of the pipeline with a high-level overview.

Q.4 a. Explain Cyrus-Beck clipping algorithm for a convex polygon with an example. (8)

Answer: Cyrus–Beck is a general algorithm and can be used with a convex polygon clipping window.

p(t) = p0 + t(p1-p0) /* it's parametric function */

3] if > 0 ; vector says p(t) is OUTSIDE && A < 90 degree.

if < 0 ; vector says p(t) is INSIDE && a > 90 degree.

if = 0 ; vector says p(t) is on edge E .. here outer normal edge is perpendicular to the E and p(t)-B

.. we will writing here a function code for it as given below :

```
/*

if( DtProd (N,P(t)-B) > 0)
      {
        p(t) OUTER & A < 90 degree ;    /* P(t) is OUTSIDE ..

      }
else if( DtProd (N,P(t)-B) < 0)
      {
        p(t) INNER & A > 90 degree ;    /* P(t) is INSIDE ..

      }
else( DtProd (N,P(t)-B) = 0)
      {
        p(t) lies on to the edge E ;    /* where outer normal edge N would be perpendicular to both E and p(t)-B..

      }
*/
```

b. Distinguish between various OpenGL point-attribute and OpenGL line attribute functions. (8)

Answer: POINT PLOTTING

The function glVertex () specifies the coordinates for a point position.
We define world-coordinate positions with glVertex functions placed between a glBegin/glEnd pair using the primitive type constant: GL_POINTS. Coordinate positions can be specified in two or three dimensions. We can also use homogeneous-coordinate representations (four dimensional). Default values for the *z* coordinate and the *h* parameter in coordinate specifications are $z = 0$ and $h = 1$. We use a suffix (2, 3, or 4) on the glVertex to indicate the coordinate dimension.

The data type to be used in specifying a particular cordinate position is also indicated with a suffix code on the glVertex function. These suffix codes are double (d), float (f), integer (i), and short (s). Coordinate values can be explicitly listed, or they can be given in a separate array designation. For an array specification of a coordinate position, we append a third suffix code: v (for "vector").

In the following example, three points are plotted along a two-dimensional straight-line path with a slope of 2. Coordinates are given as integers.

```
glBegin (GL_POINTS);
  glVertex2i (50, 100);
  glVertex2i (75, 150);
  glVertex2i (100, 200);
glEnd ();
```

Alternatively, we could have used a vector specification for coordinate positions by replacing each of the statements between the glBegin/glEnd pair with a statement of the form

```
glVertex2iv (endpointCoords1);
```

where parameter endpointCoords1 is a pointer to an array of coordinate values.

LINE FUNCTIONS

As with point plotting, straight line segments are specified with glVertex functions that are placed within glBegin/glEnd pairs. In this case however, the coordinate positions are interpreted as line endpoint positions. Straight line segments are drawn as solid lines, unless other attribute options are selected. There are three primitive types in OpenGL that we can use to generate line segments:

GL_LINES	Generates a series of unconnected line segments between each successive pair of specified endpoints. Thus, we obtain one straight line segment between the first and second coordinate points, then another line segment between the third and fourth points, and so forth up to the final pair of endpoint positions. If the number of specified endpoints is odd, the last endpoint position is ignored.
GL_LINE_STRIP	Generates a "polyline" of connected line segments between the first endpoint and the last endpoint.
GL_LINE_LOOP	Generates a series of connected line segments the same as GL_LINE_STRIP, but then adds a final line segment from the last point back to the first point specified.

Example:

```
glBegin (lineMode);
  glVertex2i (50,150);
  glVertex2i (150, 150);
  glVertex2i (150, 50);
  glVertex2i (50, 50);
glEnd ();
```

If parameter lineMode in this example is set to the value GL_LINES, we obtain two unconnected line segments that are horizontal and parallel. With GL_LINE_STRIP, we have a connected polyline with three line segments between position (50, 150) and position (50, 50). And with GL_LINE_LOOP, we draw the four edges of a square, where each edge is 100 pixels long.

b. Develop a general form of scaling matrix about a fixed point (xf, yf). (8)

Answer: Scaling (magnification or miniaturization)

When scaling an object from the point of origin by the factor s, the point (x,y) is mapped to $(x', y') = (s \cdot x, s \cdot y)$.

If we use different scaling factors sx and sy in x- respectively y-direction, we get
(x´,y´) = (sx·x, sy·y)
Scaling with respect to a point other than the origin:
1st step = translation of the scaling center into the
point of origin: T(–xf,–yf)
2nd step = scaling of the object with respect to the
point of origin: S(sx,sy)
3rd step = translation of the object back to its
original location: T–1(–xf,–yf) = T(xf,yf)
So we obtain the generalized scaling matrix with (xf,yf) as scaling center by:
S(xf,yf,sx,sy) = T(xf,yf)·S(sx,sy)·T(–xf,–yf)

Q.6 a. Define a polygonal mesh. What are the Properties of meshes? (8)
Answer:

6.2.1 To Define a Polygonal Mesh

A polygonal mesh is a collection of polygons along with a normal vector associ with each vertex of each polygon. We begin with an example.

■ **EXAMPLE 6.2.1 The basic barn**

Figure 6.5 shows a simple shape we call the basic barn. It has seven polyg faces and a total of 10 vertices (each of which is shared by three faces). For c venience it has a square floor one unit on a side. (The barn could be scaled oriented appropriately before being placed in a scene.) Because the barn is sumed to have flat walls, there are only seven distinct normal vectors involv the normal to each face as shown.

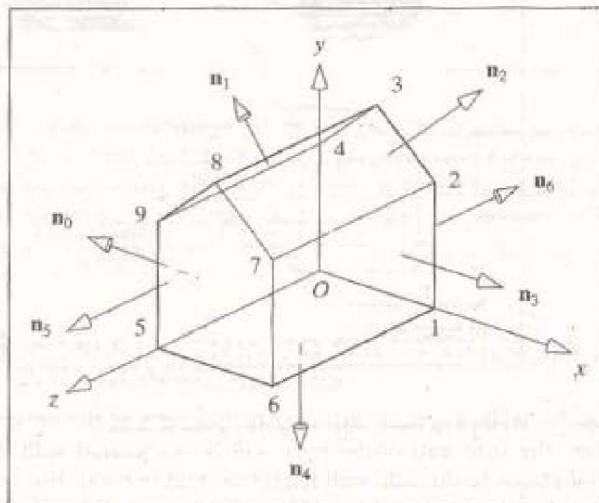


FIGURE 6.5 Introducing the basic barn.

Vertex	x	y	z
0	0	0	0
1	1	0	0
2	1	1	0
3	0.5	1.5	0
4	0	1	0
5	0	0	1
6	1	0	1
7	1	1	1
8	0.5	1.5	1
9	0	1	1

FIGURE 6.6 Vertex list for the basic barn.

There are various ways to store mesh information in a file or program. For the barn you could use a list of seven polygons, and list for each one where its vertices are located and the direction in which the normal for each of its vertices points (a total of 30 vertices and 30 normals). This would be quite redundant and bulky, however, since there are only 10 distinct vertices and seven distinct normals.

A more efficient approach uses three separate lists, a **vertex list**, **normal list**, and **face list**. The vertex list reports the locations of the distinct vertices in the mesh. The list of normals reports the directions of the distinct normal vectors that occur in the model. The face list simply indexes into the vertex and normal lists. As we see next, the barn is thereby captured with 10 vertices, seven normals, and a list of seven simple face descriptors.

The three lists work together: The vertex list contains locational or geometric information, the normal list contains **orientation** information, and the face list contains connectivity or **topological** information.

The vertex list for the barn is shown in Figure 6.6. The list of the seven distinct normals is shown in Figure 6.7. The vertices have indices 0 through 9 and the normals have indices 0 through 6. The vectors shown have already been normalized, since most shading algorithms require unit vectors. (Recall that a cosine can be found as the dot product between two unit vectors.)

Normal	n_x	n_y	n_z
0	−1	0	0
1	−0.707107	0.707107	0
2	0.707107	0.707107	0
3	1	0	0
4	0	−1	0
5	0	0	1
6	0	0	−1

FIGURE 6.7 The list of distinct normal vectors involved.

Figure 6.8 shows the barn's face list: each face has a list of vertices and the normal vector associated with each vertex. To save space, only the indices of the proper vertices and normals are used. (Since each surface is flat, all of the vertices in a face are associated with the same normal.) The list of vertices for each face begins with any vertex in the face and then proceeds around the face, vertex by vertex, until a complete circuit has been made. There are two ways to traverse a polygon: clockwise and counterclockwise. For instance, face #5 above could be listed as (5, 6, 7, 8, 9) or (9, 8, 7, 6, 5). Either direction could be used, but we follow a convention that proves handy in practice:

Traverse the polygon counterclockwise as seen from outside the object.

Using this order, if you traverse around the face by walking on the outside surface from vertex to vertex, the interior of the face is on your left. We later design algorithms that exploit this ordering. Because of it, the algorithms are able to distinguish with ease the front from the back of a face.

Face	Vertices	Associated Normal
0 (left)	0, 5, 9, 4	0, 0, 0, 0
1 (roof left)	3, 4, 9, 8	1, 1, 1, 1
2 (roof right)	2, 3, 8, 7	2, 2, 2, 2
3 (right)	1, 2, 7, 6	3, 3, 3, 3
4 (bottom)	0, 1, 6, 5	4, 4, 4, 4
5 (front)	5, 6, 7, 8, 9	5, 5, 5, 5, 5
6 (back)	0, 4, 3, 2, 1	6, 6, 6, 6, 6

FIGURE 6.8 Face list for the basic barn.

© IETE

The barn is an example of a data-intensive model, where the position of the vertex is entered by the designer. In contrast, we see later some models that are generated algorithmically. For instance, some prospective home owners may want to design the floorplan of their dream house and let a CAD software program or an architect flesh out the actual 3D contours or a full-color rendition of the house. For some buildings such as a house or hospital, or the Pentagon in Washington, it would be an enormous task to fill in the tables by hand. A likely substitute would be to have the table data stored in files. Having these files it isn't too hard to... there, it isn't too hard to come up with the vertices for the basic barn or house.

Some CAD and 3D graphics programs, such as AutoCAD and 3D Studio, automatically create lists such as those described in the previous tables. The designer of the hospital, factory, or church can save all of the data in a file with a write command and later make it available for rendering in an application.

6.2.3 Properties of Meshes

Given a mesh specified by its vertex, normal, and face lists, we might wonder what kind of an object it represents. Some properties of interest are:

- **Solidity**: As mentioned earlier, a mesh represents a solid object if its faces together enclose a positive and finite amount of space.
- **Connectedness:** A mesh is **connected** if every face shares at least one edge with some other face. (If a mesh is not connected, it is usually considered to represent more than one object.)
- **Simplicity**: A mesh is **simple** if the object it represents is solid and has no holes through it; it can be deformed into a sphere without tearing. (Note that the term simple is being used here in quite a different sense from that for a simple polygon.)
- **Planarity**: A mesh is planar if every face is a **planar** polygon: the vertices of each face then lie in a single plane. Some graphics algorithms work much more efficiently if a face is planar. Triangles are inherently planar, and some modeling software takes advantage of this by using only triangles. Quadrilaterals, on the other hand, may or may not be planar. The quadrilateral in Figure 6.10, for instance, is planar if and only if $a = 0$.
- **Convexity**: A mesh represents a **convex** object if the line connecting any two points within the object lies wholly inside the object. Convexity was first discussed in Section 2.3.6 in connection with polygons. Figure 6.11 shows some convex and some nonconvex objects. For each nonconvex object an example line is shown whose endpoints lie in the object but which is not itself contained within the object.

b. Write short notes on: (8)
(i) Stereo view
(ii) Toxonomy of Projections

Answer:

7.5 TO PRODUCE STEREO VIEWS

I love 5.1. Sometimes you can't squeeze everything in comfortably into a stereo picture. There is a lot more space in a 5.1 environment.

Peter Gabriel (1950–)

We digress briefly to use the camera controls developed earlier for producing stereo views of a scene. A stereo view can make a picture much more intelligible; when it is

ed properly the viewer obtains a sense of depth in a picture, which not only es it much more interesting and realistic, but also reduces the visual ambiguity he picture (such as which lines lie in front of others.) All of the stereo figures in s book were made using the following technique.

We might call the camera used so far a "cyclops" camera, after the one-eyed mon-er polyphemus, son of Poseidon, in greek mythology." Also, replace "fabled" with ried". To get a sense of its limitations, keep one eye closed as you walk around a m and try to do simple tasks. Our natural stereoscopic eye–brain system pro-es a tremendous amount of information by adding a visual sense of depth. We nt to add this capability to computer graphics pictures.

To make a stereo view, two pictures, a left and a right picture, are made using ghtly different cameras, as suggested in Figure 7.34. The cameras are built using same lookAt point but different eye positions. Two viewports are created side side on the display as in Figure 7.34b. The left picture is displayed in the left view-t and the right picture in the right viewport.

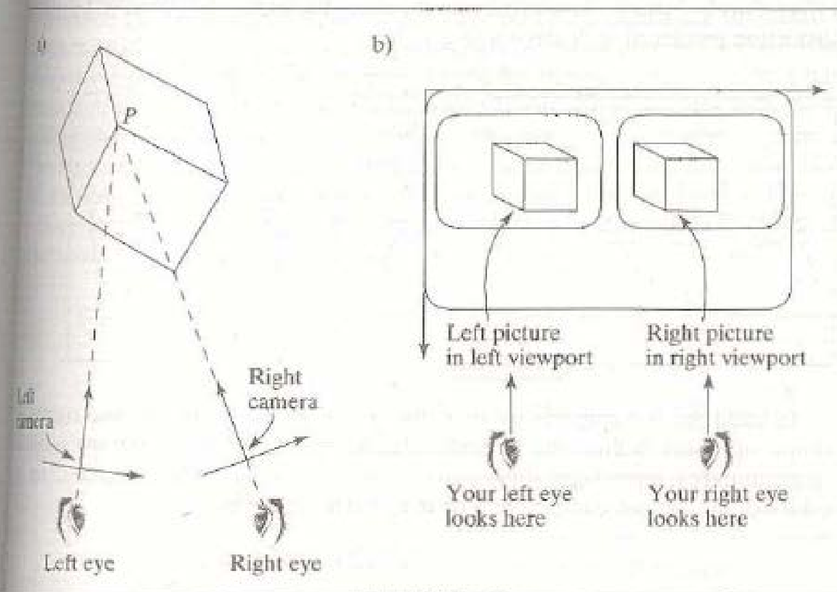


FIGURE 7.34 Creating stereo views.

To view a stereo picture, let your left eye look at the left picture and your right eye look at the right picture. When you do this properly, the two images fuse into a single image that appears to have depth. This may take some practice. (The preface describes a method for learning how to do this.)

Figure 7.35 shows a stereo wireframe view of the Buckyball described in Chapter 6. The two pictures are evidently quite different, and there is significant visual ambiguity (which edges are in front; which behind?) when only one of the pictures is viewed. A stereo view, however, disambiguates the various edges, making the picture easily intelligible.

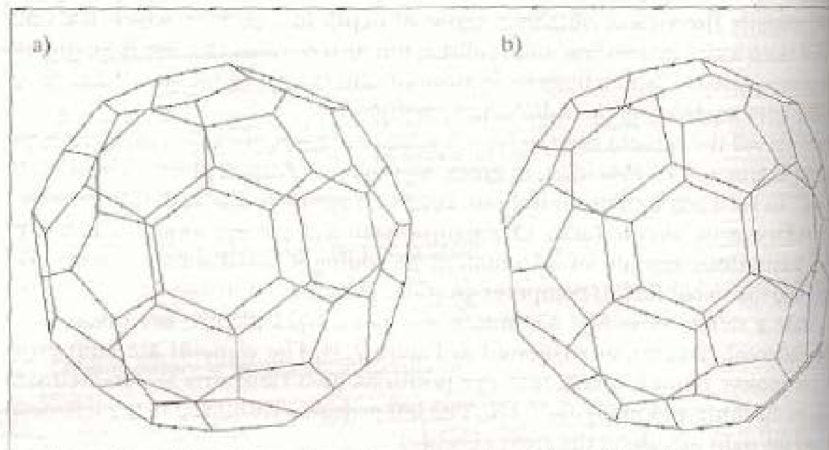


FIGURE 7.35 Stereo view of the Buckyball.

Figure 7.36 shows stereo views of the barn. In part a the camera is rolled by Note that the orientation of the barn is difficult to comprehend without the ste effect. Part b shows a close-up of one corner of the barn, and the severe perspe distortion produced is clearly visible.

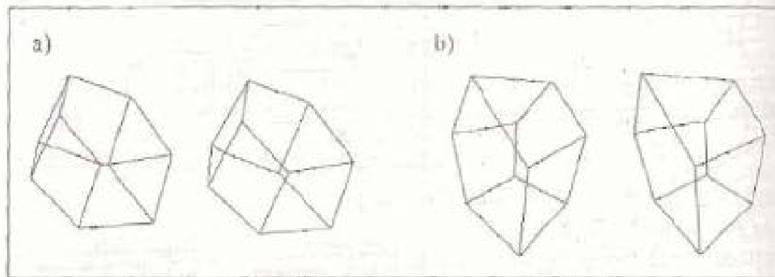


FIGURE 7.36 Close-up views of the barn.

To build the two cameras we must decide where to put the left and right eye simple approach begins with a regular camera based on a single lookAt point and single initial cyclops eye, as suggested in Figure 7.37. Along with a choice of up, the establish the cyclops camera, with its u, v, and n directions.

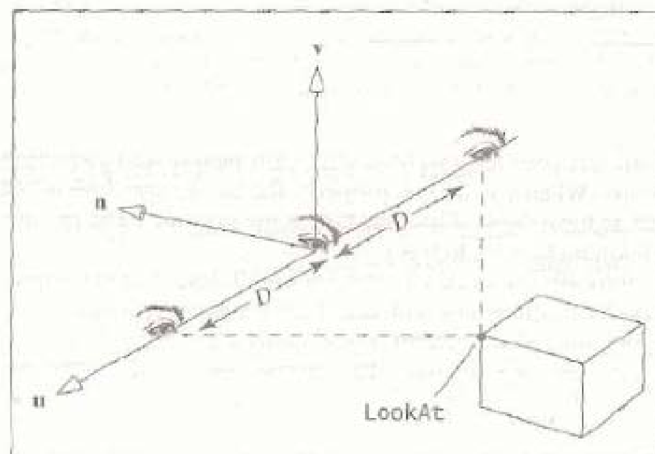


FIGURE 7.37 Setting the two eye positions for stereo viewing.

The left and right eyes are defined at slight displacements of the cyclops eye, at ne distance D along $-\mathbf{u}$ and $\mathbf{u}$, respectively. The choice of D depends on the unit measure being used in the application. If all lengths and distances are being ught of as inches, then the user will probably set up the camera at an appropriate mber of inches from the desired lookAt point. Human eyes are about 3 inches art, so a good first choice of D would be 1.5. If things were measured in meters in-ad, you might use the distance in meters between eyes. In those cases where the ne is fanciful and has no inherent scale, some experimentation would be needed achieve the desired visual effect. Case Study 7.2 suggests a project to produce reo views.

7.6 TAXONOMY OF PROJECTIONS

Lines, so loves oblique, may well Themselves in every angle greet; But ours, so truly parallel, Though infinite, can never meet.

The Definition of Love
Andrew Marvell
(1621–1678).

We examined the basic ideas of *planar projections*, where points are projected in one way or another onto a plane. We looked at parallel projections in Chapter 5, and perspective projections in this chapter. There are many special cases that have been used in art, architecture, and engineering drawings, and we now look to see what their characteristics are, and how they fit together.

Planar projections fall naturally into the tree structure shown in Figure 7.38. Each child of a projection type represents a special case of its parent in the tree. The first fundamental split is between parallel and perspective projections. We shall first examine classes of perspective projections.

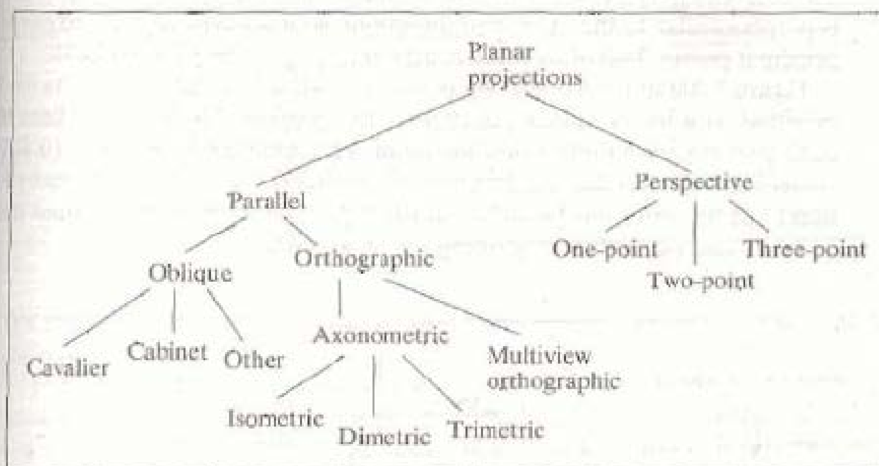


FIGURE 7.38 A taxonomy popular projections.

7.6.1 One-, Two-, and Three-Point Perspective

Perspective projections divide nicely into three classes: one-point, two-point, and three-point. They are distinguished by the orientation of the camera relative to the world coordinate system. The names derive from the situation of viewing the unit cube shown in Figure 7.39. The unit cube is nestled into the positive x-, y-, z-octant with one corner at the origin. Most important, its edges are aligned with the world coordinate axes, which in this discussion are called **principal axes**. The principal axes lie in the directions of the

unit vectors **i**, **j**, and **k**. Similarly, the three planes $x = 0$, $y = 0$, and $z = 0$ are called the **principal planes**, and the cube has its six faces aligned with them.

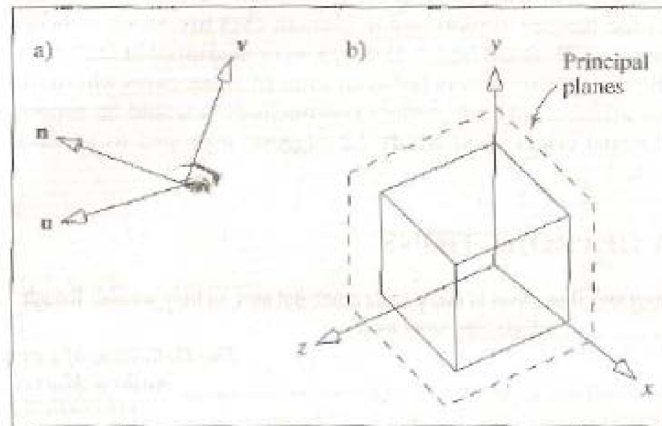


FIGURE 7.39 The unit cube, the principal axes, and the principal planes.

The camera can be oriented in an infinite number of ways relative to this coordinate system. For some of these the n-axis of the camera is perpendicular to one principal axis or another. Traditionally, perspective projections are categorized by counting the number of **finite vanishing points** that the principal axes produce. Recall that if a line is perpendicular to **n**, its vanishing point is at infinity; otherwise it is finite. So we can also count the number of principal axes that are *not* perpendicular to **n**. This is also the number of principal axes that **pierce** the viewplane of the camera. (Why?)

1. **One-point perspective**: Exactly one principal axis has a finite vanishing point. Thus **n** is not perpendicular to exactly one of the three directions **i**, **j**, or **k**. But it is perpendicular to the other two directions, so it is perpendicular to one of the principal planes. Two of its three components, n_x, n_y, or n_z, must be 0.

   Figure 7.40a shows a one-point perspective view, in which the camera has been oriented with its viewplane parallel to the xy-plane. The receding lines of the cube converge to a finite vanishing point. The camera here has $\mathbf{n} = (0, 0, 1)$. In camera coordinates the receding lines have direction $\mathbf{c} = (0, 0, -1)$, so by Equation (7.6) the vanishing point lies at $(0, 0)$. On the other hand the lines parallel to the x- and y-axes have vanishing points at infinity.

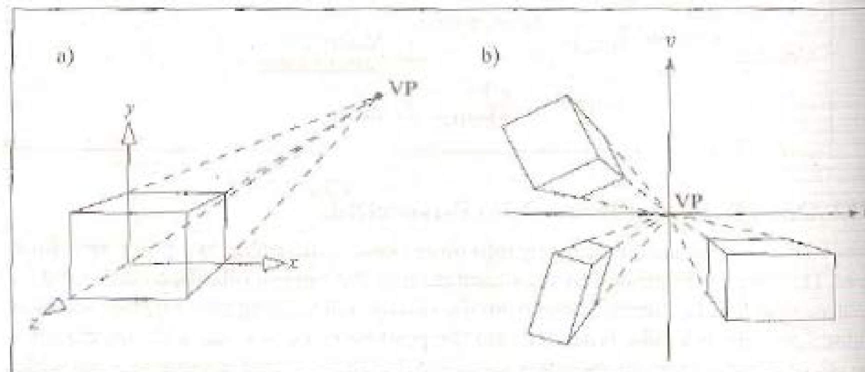


FIGURE 7.40 One-point perspective views.

© IETE

The location of the vanishing point does not depend on the position of the camera relative to a cube. Figure 7.40b shows several blocks in one-point perspective. Each block may be considered to have its own principal axes, and in this picture the front face of each block is parallel to the viewplane. All receding lines share the same vanishing point (0, 0).

Revisit Figure 7.24, which shows two sets of grid lines on a horizontal plane. The grid lines run parallel to the principal axes (the world coordinate axes). Another set of grid lines, not shown, would run vertically, parallel to the world y-axis. The figure looks like a one-point perspective, since there seems to be a single finite vanishing point at the horizon. But the camera could be aimed downward, making it a two-point perspective, as we discuss next. You can't tell from the figure alone. (If you are told that the horizon projects to $y = 0$, you can then conclude that the camera is level and that this is indeed a one-point perspective.)

2. **Two-point perspective**. Exactly two principal axes have finite vanishing points. Thus the camera's **n** direction is *not* perpendicular to two of these axes; it is perpendicular to only one. One of its three components must be 0.

Figure 7.41a shows a cube in two-point perspective: there are two finite vanishing points, since both axes **i** and **k** pierce the viewplane. The camera was set up as suggested in Figure 7.41b, with its **n** making an angle of θ with the z-axis, so that $\mathbf{n} = (\sin(\theta), 0, \cos(\theta))$. Here **n** is perpendicular to **j**, so the vertical principal axis has an infinite vanishing point.

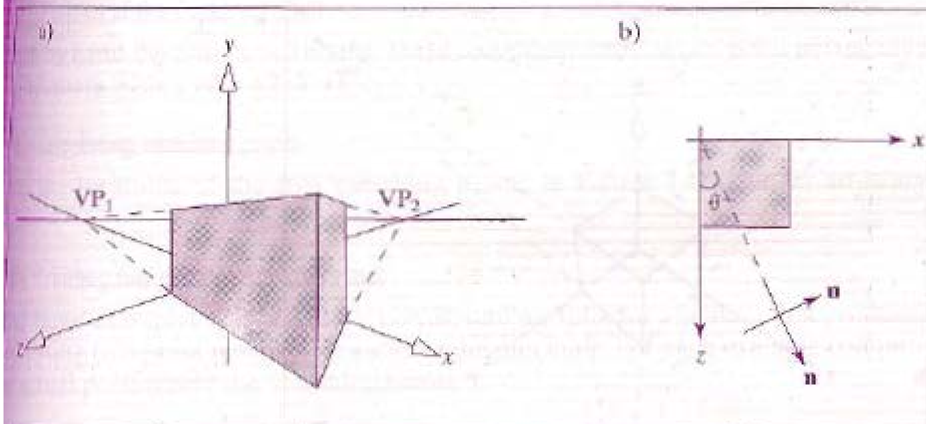


FIGURE 7.41 A two-point perspective view.

It's not hard to compute where the finite vanishing points are located (see the exercises).

It is interesting to see what happens if we view the infinite grid scene, first seen in Figure 7.24, in two-point perspective. Figure 7.42 shows the case where the eye is still at $y = 1$ oriented horizontally, but the camera has been yawed to the left so that $\mathbf{n} = (.74, 0, .67)$. Now both sets of lines recede to the horizon, producing two widely separated vanishing points on the horizon. (What are the vanishing points numerically?) Many of the more remote lines are not drawn here, as they are so crowded together that they cannot be seen clearly.

Three-point perspective. All three principal axes have finite vanishing points: all three pierce the viewplane. **n** is not perpendicular to any axes, so all of its components are nonzero.

Q.7 a. What is ambient light illumination? Write the equations for ambient light. (8)

Answer:

Ambient illumination is light that's been scattered so much by the environment that its direction is impossible to determine - it seems to come from all directions. Backlighting in a room has a large ambient component, since most of the light that reaches your eye has first bounced off many surfaces. A spotlight outdoors has a tiny ambient component; most of the light travels in the same direction, and since you're outdoors, very little of the light reaches your eye after bouncing off other objects. When ambient light strikes a surface, it's scattered equally in all directions.

Ambient Light
ambient light
Light from a diffuse, non-directional source.
The illumination of an object from ambient light can be represented by the equation:
I = Ia ka
Where:
Ia is the ambient illumination and
ka is the ambient-refelection coefficient of the object material.
ambient-reflection coefficient
A material property, the ratio of reflected light intensity to ambient light.

Ambient Sources and Ambient Reflections

To overcome the problem of totally dark shadows, we imagine that a uniform background glow called **ambient light** exists in the environment. This ambient light source is not situated at any particular place, and it spreads in all directions uniformly. The source is assigned an intensity, I_a. Each face in the model is assigned a value for its **ambient reflection coefficient,** ρ_a (often this is the same as the diffuse reflection coefficient, ρ_d), and the term $I_a\rho_a$ is simply added to whatever diffuse and specular light is reaching the eye from each point P on that face. I_a and ρ_a are usually arrived at experimentally, by trying various values and seeing what looks best. Too little ambient light makes shadows appear too deep and harsh; too much makes the picture look washed out and bland.

2.5 How to Combine Light Contributions

We can now sum the three light contributions—diffuse, specular, and ambient—to form the total amount of light I that reaches the eye from point P:

$$I = \text{ambient} + \text{diffuse} + \text{specular}$$

$$I = I_a\rho_a + I_d\rho_d \times lambert + I_{sp}\rho_s \times phong^f \qquad (8.5)$$

where we define the values

$$lambert = \max\left(0, \frac{\mathbf{s}\cdot\mathbf{m}}{|\mathbf{s}||\mathbf{m}|}\right) \quad \text{and} \quad phong = \max\left(0, \frac{\mathbf{h}\cdot\mathbf{m}}{|\mathbf{h}||\mathbf{m}|}\right) \qquad (8.6)$$

I depends on the various source intensities and reflection coefficients, as well as on the relative positions of the point P, the eye, and the point light source. Here we have given different names, I_d and I_{sp}, to the intensities of the diffuse and specular components of the light source, because OpenGL allows you to set them individually, as we see later. In practice they usually have the same value.

b. **Explain the Phong model for reflection of light from object surfaces to the viewer's eye.** **(8)**

Answer:

8.2.3 Specular Reflection

Real objects do not scatter light uniformly in all directions, and so a specular component is added to the shading model. Specular reflection causes highlights, which can add significantly to the realism of a picture when objects are shiny. In this section we discuss a simple model for the behavior of specular light due to Phong [Phong75]. It is easy to apply, and the highlights generated by Phong specular light give an object a plasticlike appearance, so the Phong model is good when you intend the object to be made of shiny plastic or glass. The Phong model is less successful with objects that are supposed to have a shiny metallic surface, although you can roughly approximate them with OpenGL by careful choices of certain color parameters. More advanced models of specular light have been developed that do a better job of modeling shiny metals. These are not supported directly by OpenGL's rendering process, so we defer a detailed discussion of them to Chapter 12 on ray tracing.

Figure 8.12a shows a situation where light from a source impinges on a surface and is reflected in different directions. In the **Phong model** we discuss here, the amount of light reflected is greatest in the direction of *perfect mirror reflection* (discussed in Chapter 4), **r**, where the angle of incidence θ equals the angle of reflection. This is the direction in which all light would travel if the surface were a perfect mirror. At other nearby angles the amount of light reflected diminishes rapidly, as indicated by the relative lengths of the reflected vectors. Part b shows this in terms of a "beam pattern" familiar in radar circles. The distance from P to the beam envelope shows the relative strength of the light scattered in that direction.

Part c shows how to quantify this beam pattern effect. We know from Chapter 4 that the direction **r** of perfect reflection depends on both **s** and the normal vector **m** to the surface, according to:

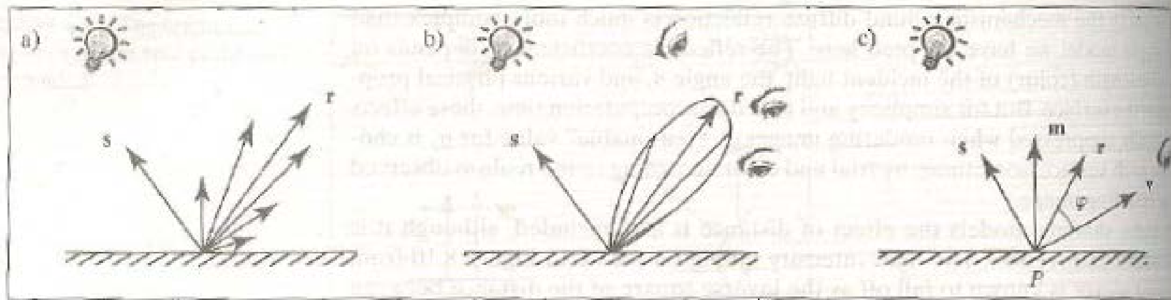


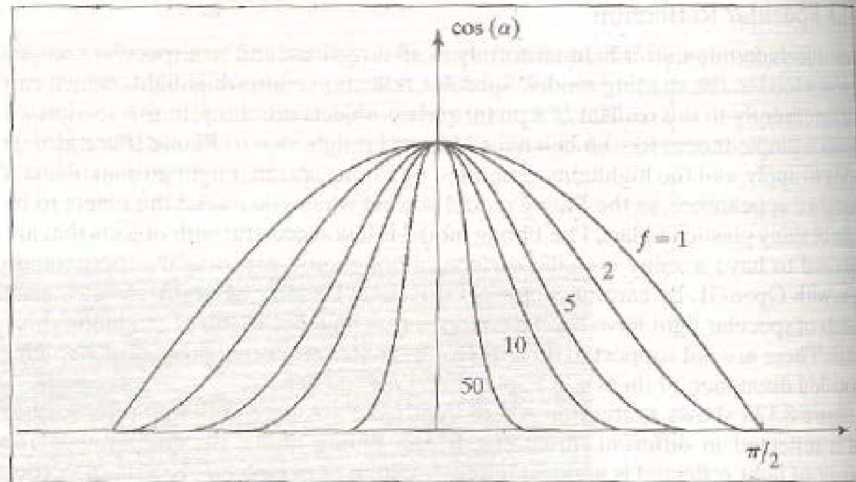
FIGURE 8.12 Specular reflection from a shiny surface.

$$\mathbf{r} = -\mathbf{s} + 2\frac{(\mathbf{s}\cdot\mathbf{m})}{|\mathbf{m}|^2}\mathbf{m} \quad \text{(the mirror-reflection direction)} \quad (8.2)$$

For surfaces that are shiny but not true mirrors, the amount of light reflected falls off as the angle ϕ between **r** and **v** increases. The actual amount of falloff is a complicated function of ϕ, but in the Phong model it is said to vary as some power f of the cosine of ϕ—that is, according to $(\cos(\phi))^f$, in which f is chosen experimentally and usually lies between 1 and 200.

Figure 8.13 shows how this intensity function varies with ϕ for different values of f. As f increases, the reflection becomes more mirrorlike and is more highly concentrated along the direction **r**. A perfect mirror could be modeled using $f = \infty$, but pure reflections are usually handled in a different manner, as described in Chapter 12.

FIGURE 8.13 The falloff of specular light with angle.



Using the equivalence of $\cos(\phi)$ and the dot product between **r** and **v** (after they are normalized), the contribution I_{sp} due to specular reflection is modeled by

$$I_{sp} = I_s\rho_s\left(\frac{\mathbf{r}}{|\mathbf{r}|}\cdot\frac{\mathbf{v}}{|\mathbf{v}|}\right)^f \quad (8.3)$$

where the new term ρ_s is the **specular reflection coefficient.** Like most other coefficients in the shading model, it is usually determined experimentally. (As with the diffuse term, if the dot product $\mathbf{r}\cdot\mathbf{v}$ is found to be negative, I_{sp} is set to zero.)

Q.8 a. How to create a new Pixmap from a combination of two pixmaps? Write an OpenGL functions for performing this operation. (8)

Answer:

There are circumstances where we wish to combine two pixmaps to produce a third. This is useful for such things as moving cursors around a screen, comparing two images, and morphing one image into another. We look at several examples of practical importance.

Pixmaps are usually combined *pixelwise*—that is, by performing some operation between corresponding pixels in the old and new pixmaps. Specifically, pixmaps A and B are combined to form pixmap C according to:

$$C[i][j] = A[i][j] \otimes B[i][j] \quad \text{for each } i, j$$

where $\otimes$ denotes some operation. Examples of different operations are:

- Averaging two images—here $\otimes$ means to form the sum of one half of A plus one half of B:

$$C[i][j] = \tfrac{1}{2}(A[i][j] + B[i][j])$$

- Differencing two images, to determine how different they are—here $\otimes$ means subtraction:

$$C[i][j] = A[i][j] - B[i][j]$$

- Finding where one image is brighter than another—here $>$ means "is greater than":

$$C[i][j] = A[i][j] > B[i][j]$$

giving each pixel in C the value 1 if the corresponding pixel in A is brighter than that in B, and 0 otherwise.

A generalization of averaging two images is to form their **weighted average**. Pixmap A is weighted by $(1 - f)$ and B is weighted by f, for some fraction f:

$$C[i][j] = (1 - f)A[i][j] + fB[i][j] \quad (9.1)$$

For instance, if the RGB components of $A[i][j]$ are (14, 246, 97) and those of $B[i][j]$ are (82, 12, 190), then for $f = 0.2$ we have $C[i][j] = (27, 199, 115)$. A weighted average of two `RGBpixmaps` can be achieved using the modification of the `setPixel()` function given in Figure 9.3.

EXAMPLE 9.3.1 Dissolving one image into another

An interesting application of a weighted average occurs when you wish to dissolve between two images. First image A is fully displayed, but as time passes A slowly fades and image B emerges superimposed on A, until finally only B is displayed. If t represents time, then at time t the image:

$$A(1 - t) + Bt$$

is displayed, as t moves smoothly from 0 to 1. This is very similar to tweening, which we described in Chapter 4. Figure 9.11 shows five stages of the displayed image, for values of $t = 0, 0.25, 0.5, 0.75$, and 1. Case Study 9.2 discusses an easy way to dissolve between two images, using the alpha channel facility of OpenGL, as we describe in Section 9.3.2.

FIGURE 9.11 Dissolving between two images.

b. What do you understand by antialiasing? Explain any two antialiasing techniques. Also write the OpenGL function to perform antialiasing. (8)

Answer:

8.1 Antialiasing Techniques

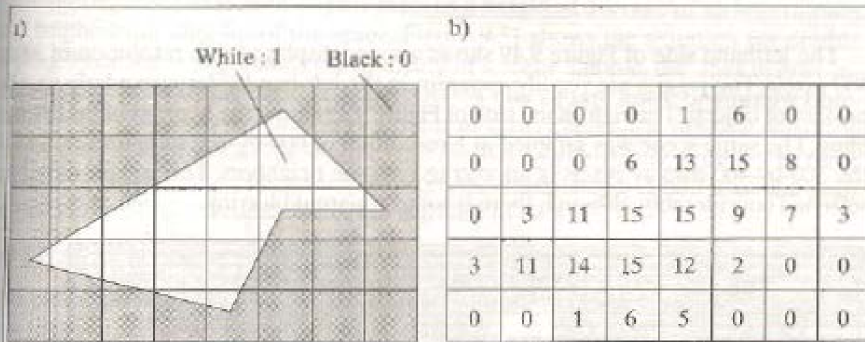
How can one reduce the aliasing produced by insufficient sampling? A higher-resolution display, coupled with better algorithms, helps, because the jags are then smaller relative to the object. But some of the jaggies still remain. We therefore look for other ways to deal with aliasing.

Antialiasing techniques involve one form or another of blurring to smooth the image. In the case of a black rectangle against a white background, the sharp transition from black to white is softened by using a mixture of gray pixels near the rectangle's border. When the picture is looked at from afar, the eye blends together the gracefully varying shades of gray and sees a smoother edge.

Three approaches to antialiasing are commonly used: **prefiltering, supersampling,** and **postfiltering**.

Prefiltering

Prefiltering techniques compute pixel colors based on an object's *coverage*: the fraction of the pixel area that is covered by the object. Consider scan-converting a white polygon in a black background, as in Figure 9.47a. Suppose the intensity values are 0 for black and 1 for white. The polygon is situated in a square grid, where the center of each square corresponds to the center of a pixel on the display. A pixel that is half-covered by the polygon should be given the intensity 1/2; one that is one-third covered should be given the intensity 1/3; and so forth. If the frame buffer has 4 bits per pixel, so that black is represented by 0 and white by 15, a pixel that is one-quarter covered by the polygon should be given the value of (1/4)15, which rounds up to 4. Figure 9.47b shows the pixel values that result when the coverage of each pixel is calculated. (What would this array of pixel values be if we instead just *sampled* the polygon at each pixel center, using level 15 when the rectangle covers the center, and 0 otherwise?)



0	0	0	0	1	6	0	0
0	0	0	6	13	15	8	0
0	3	11	15	15	9	7	3
3	11	14	15	12	2	0	0
0	0	1	6	5	0	0	0

FIGURE 9.47 Using the fraction of pixel area covered by the object.

The geometric computations required to find the coverage for each pixel can, of course, be rather time consuming. A number of efficient approaches have been developed, such as those by Pitteway and Watkinson [Pitteway80] and more recently by Xiaolin Wu [Wu91]. These algorithms calculate the coverage of each pixel in an incremental fashion, using only integer arithmetic.

In summary, prefiltering operates on the detailed geometric shape of the object(s) being scan converted and computes an average intensity for each pixel based on the objects found lying within each pixel's area. For shapes other than polygons, it can be an expensive technique computationally, and so we shall seek alternative approaches to antialiasing.

Supersampling

Since aliasing arises from sampling an object at too few points, we can try to reduce its effects by sampling more densely than one sample per pixel. This is called **supersampling**: taking more intensity samples of the scene than are displayed. Each display pixel value is formed as the average of several samples.

Figure 9.48 shows an example of double sampling: The object (in this case a tilted bar) is sampled twice more densely in both x and y than it is displayed. The squares indicate display pixels, and the x's denote spots at which the scene is sampled. Each final display pixel is formed as the average of the nine neighbor samples: the center one and the eight surrounding ones. Some samples are reused in several pixel calculations. (Which ones?) The display pixel centered at A "sees" six is based on six samples within the bar and three samples of background. Its color is set to the sum of two-thirds the bar's color and one-third the background's color. The pixel at B is based on all nine samples within the bar. Its color is set to that of the bar.

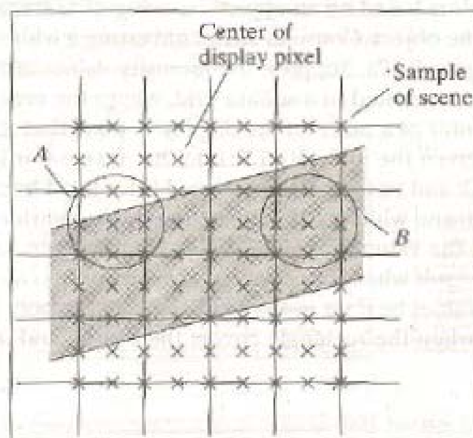


FIGURE 9.48 Antialiasing using supersampling.

The lefthand side of Figure 9.49 shows a scene displayed at a resolution of 300-by-400 pixels. The jaggies are readily apparent in the left-hand side, particularly near the profiles of objects. The right-hand side of Figure 9.49 shows the benefits of double sampling. The same scene was sampled at a resolution of 600-by-800 samples, and each of the 300-by-400 display pixels is an average of nine neighbors. The jaggies have been softened considerably, although there is some apparent blurring.

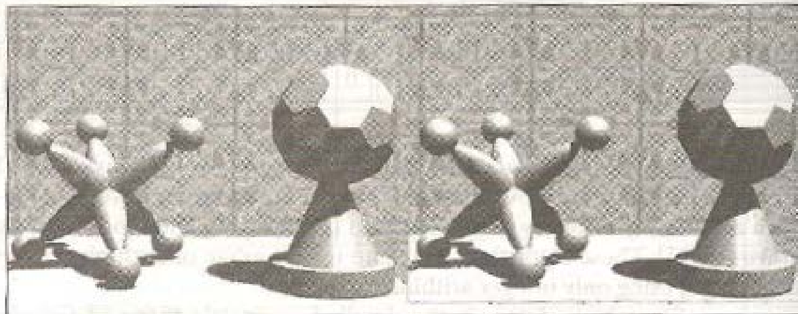


FIGURE 9.49 Objects rendered at two different sample sizes left panel without antialiasing; right panel: with double sampling.

In general, supersampling computes N_s scene samples in both x and y for each display pixel, averaging some number of neighbor samples to form each display pixel value. Supersampling with $N_s = 4$, for example, averages 16 samples for each display pixel.

e can do antialiasing even with no supersampling ($N_s = 1$). The scene is sam- t the corner of each display pixel, as suggested in Figure 9.50. The intensity of display pixel is set to the average of the four samples taken at its corners. Some ning of the jaggies is still observed, even though there is no supersampling.

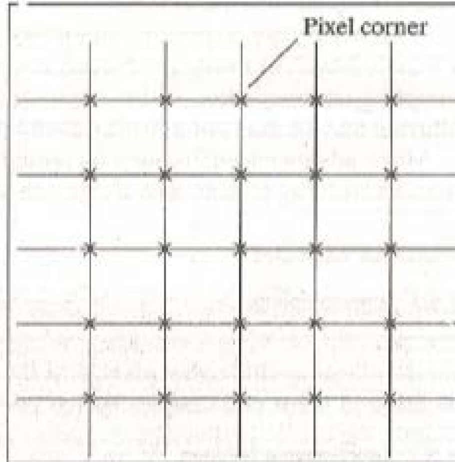


FIGURE 9.50 Antialiasing by corner sampling.

tfiltering

e double-sampling method, nine neighboring samples are averaged to compute h display pixel's intensity, giving each neighbor equal importance. This form of ning or filtering might be improved by giving the center sample more weight and eight neighbors less weight. Or it may help to include more neighbors in the av- ging computation.

Postfiltering computes each display pixel as a **weighted average** of an appropriate t of neighboring samples of the scene. Figure 9.51 shows the situation for double pling. Each value represents the intensity of a scene sample, the ones in gray in- ating the centers of the various display pixels. The square **mask** or **window func- n** of weights is laid over each gray square in turn. Then each window weight is ltiplied by its corresponding sample, and the nine products are summed to form e display pixel intensity. For example, when the mask shown is laid over the sam- e of intensity 30, the weighted average is found to be

$$(30)/2 + (28 + 16 + 4 + 42 + 17 + 53 + 60 + 62)/16 = 32.625$$

ich rounds to intensity 33. This mask gives eight times as much weight to the cen- as to the other eight neighbors. The weights always sum to 1.

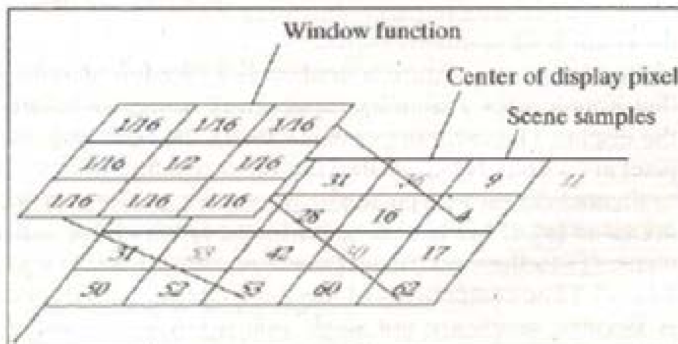


FIGURE 9.51 Postfiltering a graphics image.

Note that supersampling as we have described it is just a special case of postfiltering in which all the weights have value 1/9. Sampling and filter theory from the signal processing field provide analytical methods for determining how different classes of window functions perform as postfilters. Sometimes larger masks, 5-by-5 or even 7-by-7 are used. These look farther into the neighborhood of the center sample and can provide additional smoothing.

Postfiltering can be performed for any value of oversampling N_s. If $N_s = 4$ is used a 5-by-5, 7-by-7, or even 9-by-9 mask is appropriate. If $N_s = 1$, as in the case of common sampling, one might use a 3-by-3 mask that weights the center pixel most heavily. The blurring may or may not pay off, depending on the scene being rendered.

More advanced techniques for antialiasing are discussed in Chapter 12 in connection with ray tracing and are also discussed on line.

9.8.3 Antialiasing Using OpenGL

OpenGL provides some tools to perform antialiasing. The simplest to use employs an **accumulation buffer**, which is an extra storage area similar to the frame buffer that OpenGL can create and draw into. The antialiasing method resembles stochastic sampling. It draws a scene multiple times at slightly different positions (which differ by just fractions of a pixel) and adds the results into the accumulation buffer. When all of the slightly perturbed drawings have been added to the accumulation buffer, the results are copied over into the frame buffer and the antialiased drawing is displayed. Thus the method forms in each pixel an average value based on colors in the projected scene that lie in the immediate vicinity of the pixel.

The following code shows how an example of how this can be done when a camera is taking a picture of a 3D scene. The accumulation buffer is created at startup[5] and initially zeroed out (using `glClear(GL_ACCUM_BUFFER_BIT)`. Then the scene is drawn eight times, each time translating the camera in x and y (recall Chapter 5 and 7) by a small displacement stored in an array `jitter[]` of vectors. Each new drawing is scaled by 1/8 and added pixel by pixel to the accumulation buffer using `glAccum(GL_ACCUM, 1/8.0)`. When the eight renditions have been drawn, the accumulation buffer is copied into the frame buffer using `glAccum(GL_RETURN, 1.0)`.

```
glClear(GL_ACCUM_BUFFER_BIT); // clear the accumulation buffer
for(int i=0; i < 8; i++)
{
    cam.slide(f * jitter[i].x, f * jitter[i].y,0); // slide the
                                                        camera
    display(); // draw the scene
    glAccum(GL_ACCUM, 1/8.0); // add to the accumulation buffer
}
glAccum(GL_RETURN, 1.0); // copy accumulation buffer into frame
                          buffer
```

Q.9 a. Define Bezier Curve. Explain the properties of Bezier Curve. (8)

Answer:

10.5 PROPERTIES OF BEZIER CURVES

A little inaccuracy sometimes saves a ton of explanation.

H. H. Munro (Saki)
(1870–1916)

Bezier curves have some important properties that make them well suited for CAD. We will find later that these properties apply to B-splines as well. Exploring these properties and their proofs provides a great deal of insight into Bezier curves.

Endpoint Interpolation

The Bezier curve $P(t)$ based on control points $P_0, P_1, \ldots, P_L$ does not generally pass through, or interpolate, all of the control points. But we have seen that it always does interpolate P_0 and P_L. This is a very useful property, because a designer who is inputting a sequence of points thereby knows precisely where the Bezier curve will begin and end.

Affine Invariance

It is often necessary to subject a Bezier curve to an affine transformation in order to scale it, orient it, or position it for subsequent use. Suppose we wish to transform point $P(t)$ on the Bezier curve of Equation (10.25) to the new point $Q(t)$, using the affine transformation T. (T is represented by a 3-by-3 matrix in the 2D case and by a 4-by-4 matrix in the 3D case.) So $Q(t) = T(P(t))$. It appears that to find $Q(t)$ at any given value of t we must first evaluate $P(t)$, and then transform it, effectively starting over fresh for each new t. But this isn't so. We need only transform the control points (once), and then use these new control points in the same Bernstein form to re-create the transformed Bezier curve at any t! That is:

$$Q(t) = \sum_{k=0}^{L} T(P_k) B_k^L(t) \quad (10.30)$$

Affine invariance means that the transformed curve is identical to the curve based on the transformed control points.

Figure 10.16 shows a Bezier curve based on four control points $P_0, \ldots, P_3$. These points are rotated, scaled, and translated to the new control points Q_k, and the Bezier curve determined by them is drawn. This curve is identical point by point to the result of transforming the original Bezier curve.

Bezier curves are affine invariant for a very simple reason: they are formed as an affine combination of points, and from Section 5.2 we know that an affine transformation preserves affine combinations.

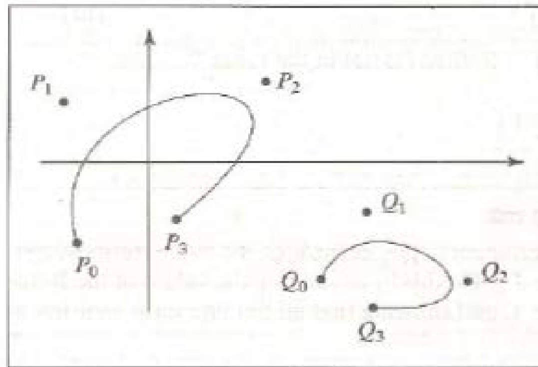


FIGURE 10.16 Showing affine invariance.

Convex Hull Property

Another property that designers may rely on is that a Bezier curve, $P(t)$, never wanders outside its convex hull. Recall from Chapter 5 that the convex hull of a set of points $P_0, P_1, \ldots, P_L$ is the set of all *convex combinations* of the points—that is, the set of all points given by

$$\sum_{k=0}^{L} \alpha_k P_k \qquad (10.31)$$

where each α_k is nonnegative, and they sum to 1.

But $P(t)$ of Equation (10.33) *is* a convex combination of its control points for every t, since no Bernstein polynomial is ever negative, and they sum to 1. Thus every point on the Bezier curve is a convex combination of its control points, so it must lie within the convex hull of the control points.

The convex hull property also follows immediately from the fact that each point on the curve is the result of tweening two points that are themselves tweens, and the tweening of two points forms a convex combination of them. Figure 10.17 illustrates how the designer can use the convex hull property. Even though the eight control points form a jagged control polygon, the designer knows the Bezier curve will flow smoothly between the two endpoints, never extending outside the convex hull.

Derivatives of Bezier Curves

Because a curve can exhibit corners and other abrupt changes when its derivatives with respect to t have discontinuities, we must investigate the various derivatives of $P(t)$ in Equation (10.25).

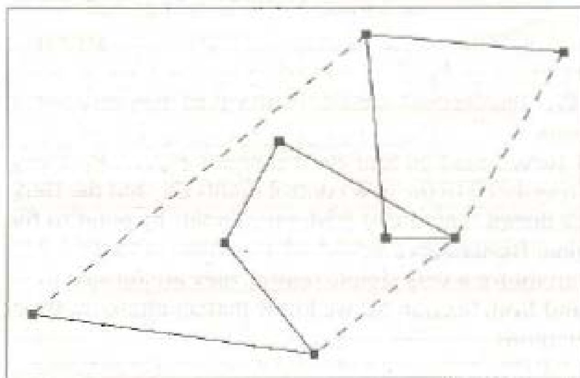


FIGURE 10.17 Using the convex hull property.

For a Bezier curve one can show that the first derivative is

$$\mathbf{p}'(t) = L\sum_{k=0}^{L-1} \Delta P_k B_k^{L-1}(t) \qquad (10.32)$$

where

$$\Delta P_k = P_{k+1} - P_k \qquad (10.33)$$

(See the exercises.) So the velocity is another Bezier curve, built on a new set of control vectors ΔP_k. We simply *difference* the original control points, $\Delta P_k = P_{k+1} - P_k$, in pairs to form the control vectors of the velocity. Note from the form $B_k^{L-1}(t)$ that taking the derivative lowers the order of the curve by 1. For instance, the derivative of a cubic Bezier curve is a quadratic Bezier curve. The smoothness of Bezier curves is addressed in the exercises.

b. Discuss the de Casteljau algorithm to any number of points to generate a Bezier Curve. (8)

Answer:

Extending the de Casteljau Algorithm to Any Number of Points

We have seen that the de Casteljau algorithm uses tweening to produce quadratic parametric representations when three points are used, and cubic representations when four points are used. It generalizes gracefully to the case in which $L + 1$ control points $P_0, P_1, \ldots, P_L$ are used. For each value of t a succession of generations are built up, each one by tweening adjacent points produced in the previous generation:

$$\mathbf{P}_i^1(t) = (1 - t)\mathbf{P}_i^0(t) + t\mathbf{P}_{i+1}^0(t)$$

$$\ldots$$

$$\mathbf{P}_i^L(t) = (1 - t)\mathbf{P}_i^{L-1}(t) + t\mathbf{P}_{i+1}^{L-1}(t) \qquad (10.24)$$

for $i = 0, \ldots, L$. The superscript k in $\mathbf{P}_i^k(t)$ denotes the generation. The process starts with $\mathbf{P}_i^0(t) = P_i$ and ends with the final Bezier curve $P(t) = P_0^L(t)$. The resulting Bezier curve can be written in terms of Bernstein polynomials

$$P(t) = \sum_{k=0}^{L} P_k B_k^L(t) \qquad (10.25)$$

where the kth Bernstein polynomial of degree L is defined as[3]

$$B_k^L(t) = \binom{L}{k}(1 - t)^{L-k}t^k \qquad (10.26)$$

Here $\binom{L}{k}$ is the **binomial coefficient function**, given by

$$\binom{L}{k} = \frac{L!}{k!(L - k)!} \qquad \text{for } L <= k \qquad (10.27)$$

The value of this term is 0 if $L < k$. Each of the Bernstein polynomials is seen to be of degree L. As before, the Bernstein polynomials are the terms one gets when expanding $[(1 - t) + t]^L$, so we are assured that

$$\sum_{k=0}^{L} B_k^L(t) = 1 \qquad \text{for all } t \qquad (10.28)$$

and that $P(t)$ is a legitimate affine combination of points.

TEXTBOOK

I. **Computer Graphics Using OpenGL, F.S. Hill, Jr., Second edition, PHI/Pearson Education, 2005**

© IETE