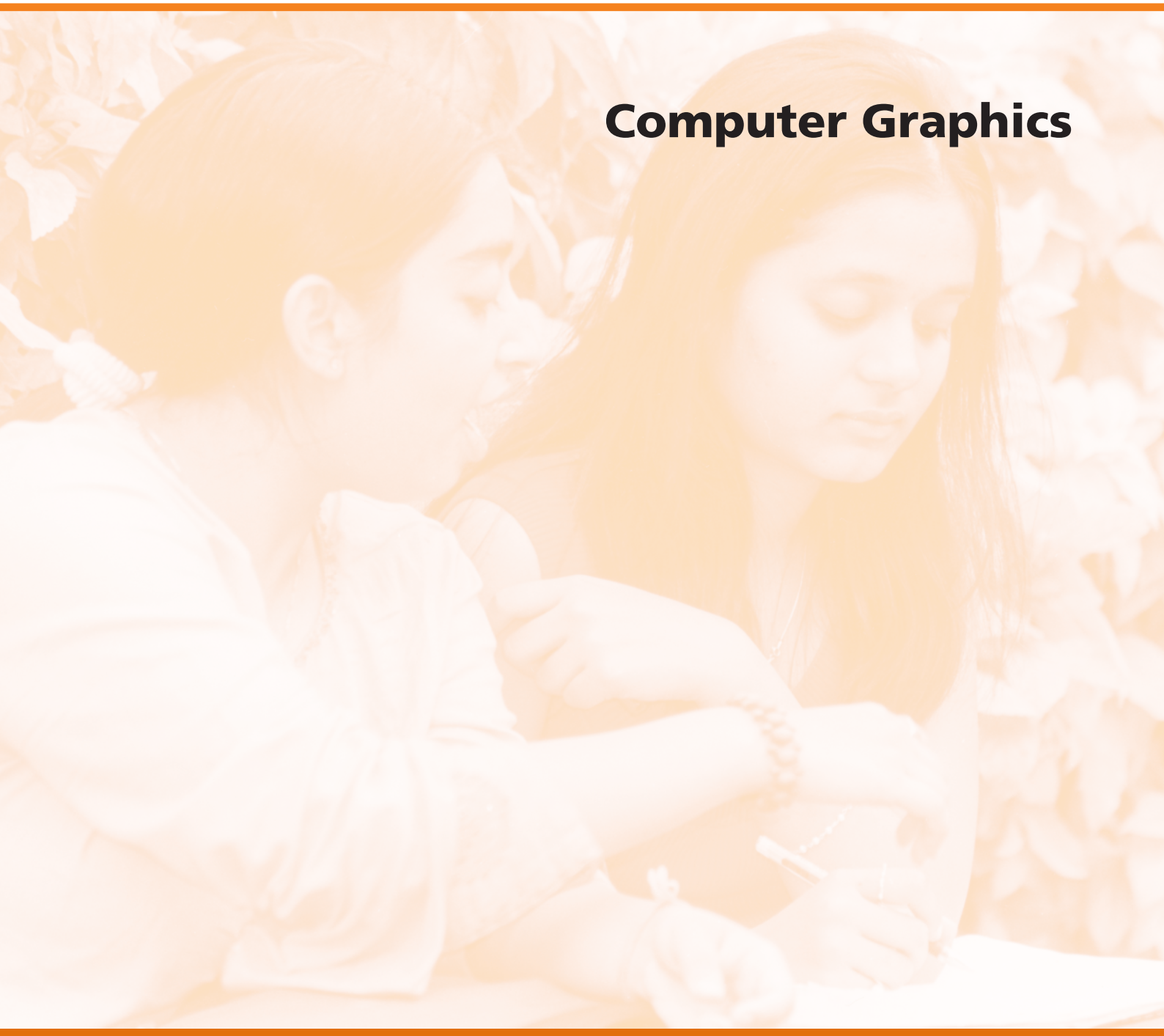
EIILM UNIVERSITY
SIKKIM

Computer Graphics



www.eiilmuniversity.ac.in

Subject: COMPUTER GRAPHICS **Credits: 4**

SYLLABUS

Introduction and Image Representation

Introduction to computer graphics and its applications, Interactive graphics, Raster graphics, Basic raster graphics algorithms for drawing 2D primitives -Scan conversion algorithms of line circle and ellipse

Graphics Hardware

Graphic display devices - Hardcopy technologies, display technologies, raster and random- scan display systems, Video controller

Clipping

Windowing and clipping - Clipping in a raster world, Clipping lines, Cyrus beck and Cohen - Sutherland Algorithms

Two-dimensional Transformations and Viewing

20 Geometrical Transformations, Matrix representations, Composition of 20 transformations, Window-to-Viewport transformation, Matrix representation of 3D Geometrical transformations, viewing in 3D Projection Introduction to Projections, Perspective projections, Parallel projections.

Geometric Representations

Hidden line/ surface removal method - Z-buffer algorithm, Introduction to Ray-tracing Illumination and Shading: Illumination models

Curves and Models

Curves and Surfaces: Polygon meshes, Parametric Cubic curves, Hermite curves, Bezier curves, Animation: Introduction and Key frame animation.

Suggested Readings:

1. James D. Foley, Andries van Dam, Steven K. Feiner, John Hughes, Computer Graphics: Principles and Practice, Addison-Wesley Professional

2. Donald Hearn, M. Pauline Baker, Computer Graphics, Prentice-Hall

3. D. P. Mukherjee, Debashish Jana, Computer Graphics: Algorithms and Implementations, PHI Learning Pvt. Ltd.

4. Krishnamurthy, Introduction to Computer Graphics, Tata McGraw-Hill Education

COURSE OVERVIEW

What is this course About?

The primary goal of computer graphics is to introduce many important data structures and algorithms that are useful for presenting data visually on a computer, computer graphics does not cover the use of *graphics design* applications such as Photoshop and AutoCAD. Nor, does it focus on the various graphics programming interfaces or graphics languages such as OpenGL or Renderman. In short, computer graphics is a programming class. The goal of this class is to provide you with sufficient background to write computer graphics applications.

Roughly speaking, the first half of this course will address a broad range of topics that that we refer to as *Raster Methods*. These include introduction to graphics, interactive building blocks, Basic raster graphic algorithms, scan conversion algorithm, two dimensional computer graphics, raster operations, Transformation, geometrical representation, matrices, Clipping algorithms. The second half of the course will cover topics related to three-dimensional computer graphics, including 3d representation, illumination, shading, visibility determination, projection techniques, raytracing and animation concepts.

This is a lot of stuff to cover in 36 class meetings, and you can expect the pace to be frantic at times. But, I am sure that you will find that computer graphics is a blast.

What do I need to know before I take this course?

You must be familiar with elementary algebra, geometry, trigonometry, and elementary calculus. Some exposure to vectors and matrices is useful, but not essential, as vector and matrix techniques will be introduced in the context of graphics as needed.

You must have at least an experience of writing computer programs in C, C++, or Java. Ideally, you have taken Computer Graphics or equivalent. We will mostly use C/C++ throughout the course, but much material will be familiar to someone whose computer language background is Java.

You need to be familiar with programming constructs such as memory allocation, reading and writing of files, pointers, structures, pointers to functions, or object oriented programming. We also assume that you can read the C/C++ code fragments that are handed out without problems. For any basic programming related questions we refer you to a textbook on C/C++.

Objectives

By the end of semester, students should:

- Have an understanding of the computer graphics its application ,interactive building block.
- Have an understanding of the operation of graphics hardware devices and software used.
- Have experience with implementing 2D graphics algorithms including scan conversion, clipping , transformation , representation , matrices.
- Have experience with implementing 3D graphics algorithms including hidden surface removal, ray tracing , representation , matrices ,projection
- Have knowledge of the major application areas of computer Animation including key frame animations and tricks.

COMPUTER GAPHICS

CONTENT

COMPUTER GRAPHICS

CONTENT

LESSON 1
AN OVERVIEW OF COMPUTER GRAPHICS, APPLICATION AND INTERACTIVE GRAPHICS

Topics Covered in the Unit

- **Introduction to computer graphics**
- **Its application**
- **Interactive graphics and their building blocks**
- **Raster graphics.**
- **Basic raster graphics algorithms for drawing 2d primitives.**
- **Scan conversion algorithm of line .**
- **Scan conversion algorithm of circle**
- **Scan conversion algorithm of ellipse**

Today's Topics

- Introduction to computer graphics
- Its application
- Interactive graphics and their building blocks

Learning Objectives

Upon completion of this chapter, the student will be able too :

- Explain what is computer graphics
- Explain its application
- Explain the interactive graphics
- Explain the interactive building blocks

Introduction

Definition: The use of a computer to create images

The term **computer graphics** includes almost everything on computers that is not text or sound. Today almost every computer can do some graphics, and people have even come to expect to control their computer through icons and pictures rather than just by typing.

Here in our lab at the Program of Computer Graphics, we think of computer graphics as drawing pictures on computers, also called rendering. The pictures can be photographs, drawings, movies, or simulations - pictures of things which do not yet exist and maybe could never exist. Or they may be pictures from places we cannot see directly, such as medical images from inside your body.

We spend much of our time improving the way computer pictures can simulate real world scenes. We want images on computers to not just look more realistic, but also to be more realistic in their colors, the way objects and rooms are lighted, and the way different materials appear. We call this work "realistic image synthesis".

Applications

Classification of Applications

The diverse uses of computer graphics listed in the previous section differ in a variety of ways, and a number of classification is by type (dimensionality) of the object to be represented and the kind of picture to be produced. The range of possible combinations is indicated in Table 1.1.

Some of the objects represented graphically are clearly abstract, some are real; similarly, the pictures can be purely symbolic (a simple 20 graph) or realistic (a rendition of a still life). The same object can of course, be represented in a variety of ways. For example, an electronic printed circuit board populated with integrated circuits can be portrayed by many different 20 symbolic representations or by 30 synthetic photographs of the board.

The second classification is by the type of interaction, which determines the user's degree of control over the object and its image. The range here includes offline plotting, with a predefined database produced by other application programs or digitized from physical models; interactive plotting, in which the user controls iterations of "supply some parameters, plot, alter parameters, report"; predefining or calculating the object and flying around it in real time under user control, as in real-time animation systems used for scientific visualization and flight simulators; and interactive designing, in which the user starts with a blank screen, defines new objects (typically by assembling them from predefined components), and then moves around to get a desired view.

The third classification is by the role of the picture, or the degree to which the picture is an end in itself or is merely means to an end. In cartography, drafting, raster painting, animation, and artwork, for example, the drawing is the end product; in many CAD applications, however, the drawing is merely a representation of the geometric properties of the object being designed or analyzed. Here the drawing or construction phase' is an important but small part of a larger process, the goal of which is to create and post-process a common database using an integrated suite of application programs.

A good example of graphics in CAD is the creation of a VLSI chip. The engineer makes a preliminary chip design using a CAD package. Once all the gates are laid out, she then subjects the chip to hours of simulated use. From the first run, for instance, she learns that the chip works only at clock speeds above 80 nanoseconds (ns). Since the target clock speed of the machine is 50 ns, the engineer calls up the initial layout and redesigns a portion of the logic to reduce its number of stages. On the second simulation run, she learns that the chip will not work at speeds below 60 ns. Once again, she calls up the drawing and redesigns a portion of the chip. Once the 'chip passes all the simulation tests, she invokes a postprocessor to create a database of information for the manufacturer about design and materials specifications, such as conductor path routing and assembly drawings. In this

Classification of Computer Graphics by Object and Picture

Type of Object
2D

3D

Pictorial Representation
Line drawing

Gray scale image

Color image

Line drawing (or *wire-frame)*

Line drawing, with various effects Shaded, color image with various effects

Interactive Graphics

Defining Interactive Computer Graphics

Computer graphics includes the process and outcomes associated with using computer technology to convert created or collected data into visual representations. The computer graphics field is motivated by the general need for interactive graphical user interfaces that support mouse, windows and widget functions. Other sources of inspiration include digital media technologies, scientific visualization, virtual reality, arts and entertainment. Computer graphics encompasses scientific, art, and engineering functions. Mathematical relationships or Geometry define many of the components in a particular computer graphics "scene" or composition. Physics fundamentals are the basis for lighting a scene. The layout, color, texture and harmony of a particular composition are established with Art and Perception principles. Computer graphics hardware and software are rooted in the Engineering fields. The successful integration of these concepts allows for the effective implementation of interactive and three-dimensional (3D) computer graphics.

Interactive 3D graphics provides the capability to produce moving pictures or animation. This is especially helpful when exploring time varying phenonmena such as weather changes in the atmosphere, the deflection of an airplane wing in flight, or telecommunications usage patterns. Interaction provides individual users the ability to control parameters like the speed of animations and the geometric relationship between the objects in a scene to one another.

The Building Blocks of Interactive Computer Graphics

Defining Primitives

Primitives are the basic geometrical shapes used to construct computer graphics scenes and the resulting final images. Each primitive has attributes like size, color, line and width. For two dimensions, examples of primitives include: a line, circle, ellipse, arc, text, polyline, polygon, and spline.

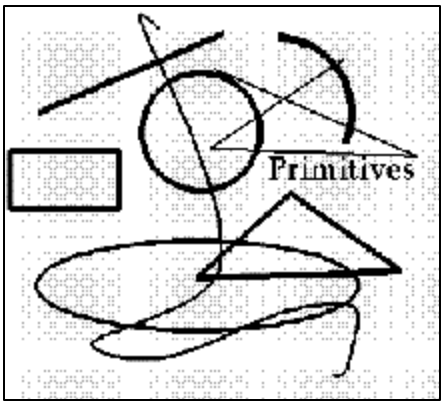


Figure #1: Examples of 2D primitives, image by Theresa-Marie Rhyne, 1997.

For 3D space, examples of primitives include a cylinder, sphere, cube and cone. 3D viewing is complicated by the fact that computer display devices are only 2D. Projections resolve this issue by transforming 3D objects into 2D displays.

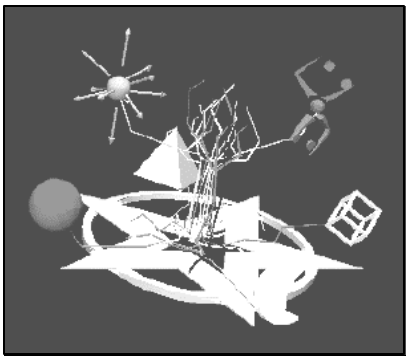
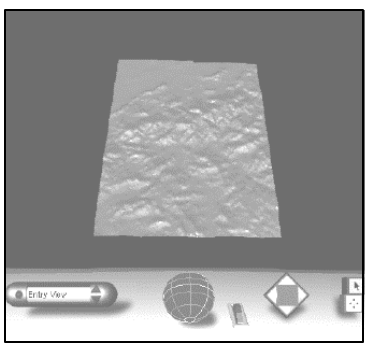


Figure #2: Examples of 3D primitives, image provided courtesy of Mark Pesce. This is a snapshot from a 3D VRML world entitled "Zero Circle".

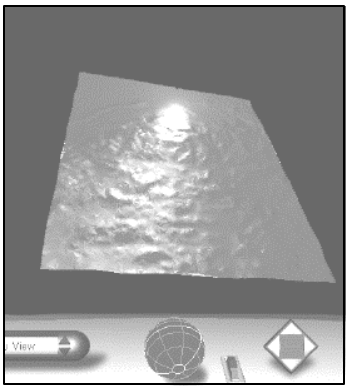
Comprehending Lighting Models

Lighting models also assist with viewing 3D volumes that are transformed on to 2D displays. For example, a 3D red ball looks like a 2D red circle if there are not highlights or shading to indicate depth. The easiest type of light to simulate is "Ambient" light. This lighting model produces a constant illumination on all surfaces of a 3D object, regardless of their orientation.

"Directional" light refers to the use of a point light source that is located at infinity in the user's world coordinates. Rays for this light source are parallel so the direction of the light is the same for all objects regardless of their position in the scene.



"Positional" light involves locating a light source relatively close to the viewing area. Rays from these light sources are not parallel. Thus, the position of an object with respect to a point light source affects the angle at which this light strikes the object.



Figures #3 & 4: Examples of Direct (Figure #3) and Positional (Figure #4) lighting. These images are courtesy of Thomas Fowler and Theresa-Marie Rhyne, (developed by Lockheed Martin for the U.S. Environmental Protection Agency),.

Understanding Color

Color is an important part of creating 3D computer graphics images. The color space or model for computer (CRT) screens is "additive color" and is based on red, green and blue (RGB) lights as the primary colors. Other colors are produced by combining the primary colors together. Color maps and pallettes are created to assist with color selection. When layering images on top of one another, the luminance equation helps determine good contrast.

So, a yellow image on a black background has very high contrast while a yellow image on a white background has very little contrast.

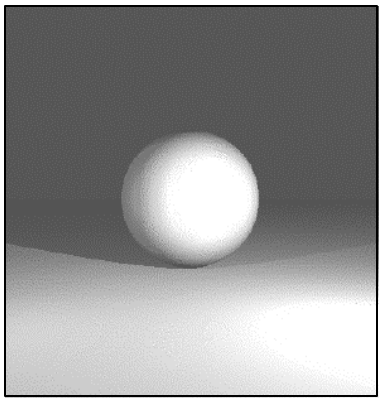


Figure #5: Example of the effects of color lights on a sphere. Image courtesy of Wade Stiell and Philip Sansone, (students in Nan Schaller's Computer Graphics course at the Rochester Institute of Technology).

The color space or model for printing images to paper is "subtractive color" and is based on cyan, magenta and yellow (CMY) as the primary colors. Printers frequently include black ink as a fourth component to improve the speed of drying inked images on paper. It is difficult to transfer from RGB color space to CMY color space. So, images on a color CRT screen can look different printed to paper.

Summary

To summarize this lesson, let's recall what we have covered? We have seen a simple definition of graphics . we have seen about the applications and their classifications, interactive graphics , various interactive building blocks of graphics ,some examples to demonstrate all these terminology. However, in the next lesson, we will work more on basic raster graphics and some of their algorithms .

Questions

1. what is computer graphics .why it is important?
2. what are the applications of computer graphics?
3. Nominate an application of computers that can be accommodated by either textual or graphical computer output. Explain when and why graphics output would be more appropriate in this application.
4. explain briefly the classification of computer graphics?

Notes:

LESSON 2
RASTER GRAPHICS, BASIC RASTER GRAPHICS ALGORITHM, SCAN CONVERSION ALGO, OF LINE, CIRCLE ELLIPSE

Today's Topics

- **Raster graphics.**
- **Basic raster graphics algorithms for drawing 2d primitives.**
- **Scan conversion algorithm of line .**
- **Scan conversion algorithm of circle**
- **Scan conversion algorithm of ellipse**

Learning Objectives

On completion of this chapter, the student will be able to :

- Explain what is raster graphics
- Demonstrate an understanding of the elementary algorithms for drawing primitive graphics shapes by programming them into a windows based program .
- Explain Scan conversion algorithms of line ,circle and ellipse.

Raster

The method used to create a graphic makes a big difference in the graphic's appearance and in its appropriateness for certain uses. This help page discusses the two most common ways to create graphics with a computer -raster and vector - and offers some guidelines on their use.

Raster, or bitmapped, graphics produce images as grids of individually defined pixels while vector graphics produce images using mathematically generated points, lines, and shapes. Until recently, raster ruled the roost on the World Wide Web, but the growing popularity of new vector-based web design programs such as Macromedia Flash is changing that. Currently, most web browsers require special plug-ins or external viewers to handle vector graphics, but a new format, SVG, promises to bring vector graphics to ordinary web pages soon. The W3C, the organization that sets standards for HTML, has standards for the SVG (scalable vector graphic) format, and popular browsers such as Netscape Navigator and Internet Explorer are have plug-ins that can allow readers to use SVG graphics.

So, how do you decide which method is appropriate for your graphics project? Here are some thoughts:

Raster

Until most browsers support the SVG format, raster is the way to go for web graphics. The dominant GIF and JPEG formats, and the increasingly popular PNG format, are raster graphics. Moreover, raster will remain the best mode for working with photographs and other images with complex and subtle shading.

Vector

The clean lines and smooth curves of vector graphics make them an obvious choice for logos, drawings, and other artwork that will appear in print. Also, the use of mathematical equations enables vector graphics to be scaled and manipulated repeatedly without distortion. The SVG format promises to deliver the same advantages to artwork on the web. In addition, vector artwork should require a lot less bandwidth than similar artwork in GIF format.

The fundamental algorithms for the development of the functions for drawing 2D primitives are introduced:

- The basic incremental algorithm
- Midpoint line algorithm
- Midpoint circle algorithm.

The Basic Algorithm

The Basic Incremental Algorithm uses the simple geometry of a line to plot points on the screen to render a line between two points. The function SetPixel() is a fictitious function. In this algorithm we assume that SetPixel() when used would draw a single pixel on the screen at the *x* and *y* coordinates given.

```
void line(int x0,int y0,int x1, int y1)
{ // for slopes between -1 and 1
int x;
float m, y;
m = (float) (y1-y0)/(x1-x0);
x=x0;
y=y0;
while ( x < x1 + 1)
{
// draw a pixel
SetPixel(x, (int) (y + 0.5));
x++;
y+=m; /* next pixel's position */
}
}
```

Midpoint Line Algorithm

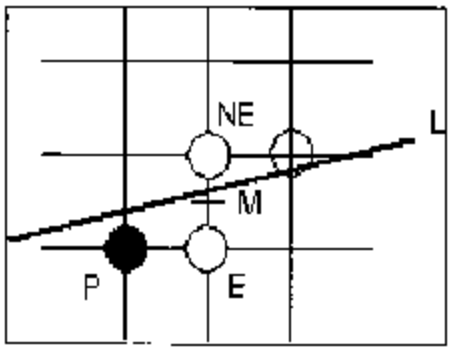
Computer graphics is very expensive in computer memory and processing. Anyone who has owned a Nintendo, Playstation or XBox will know that the computer graphics rendered by these devices needs to be of a high standard, but also generated quickly. There is always a compromise. The better the graphics, the more computing power needed. This is why the graphics rendered by today's devices is a far cry from the original games of Pong and the Commodore 64 consoles. It isn't because we never could figure out how to do it until now.

One way of making computer graphics fast to draw is to reduce the number of floating point operations in algorithms. As you can see the basic incremental line algorithm uses floating point division. This is expensive in computer processing. Therefore

other algorithms have been devised that perform scan conversions that use only integer operations. The midpoint line algorithm (otherwise known as the Bresenham Line Algorithm) is one of these.

Bresenham Line Algorithm Principals

- If point P(x , y) is in the scan-conversion, the next point is either $P(x+1, y)$ or $P(x+1, y+1)$:
- $P(x+1, y)$ is called an E-move
- $P(x+1, y+1)$ is called a NE-move
- To decide which point, use the relative position of the midpoint $M = (x+1, y+1/2)$ with respect to the line L
- The distance d can be computed incrementally with only one or two integer adds per loop!



In detail:

Next Move

- If $d < 0$, E move: $x \rightarrow x + 1$
- If $d > 0$, NE move: $x \rightarrow x + 1,\ y \rightarrow y + 1$

If $d = 0$, E or NE are equally bad: pick one

Initial Value of d

- $d_{start} = F(x_{start}+1,\ y_{start}+1/2) = F(x_0, y_0) + dy - dx/2$
- x_0, y_0 is on line so we know $F(x_0, y_0) = 0$

  $d_{start} = dy - dx/2$

To compute d with integer arithmetic, we use $F'(x, y) = 2*F(x, y)$:

- $d_{start} = 2*dy - dx$
- if E move: $d_{new} = d_{old} + 2*dy$

if NE move: $d_{new} = d_{old} + 2*dy - 2*dx$

The Code

```
void line(int x0,int y0,int x1,int y1)
{
int dx = x1 - x0,
dy = y1 - y0,
incrE,
incrNE,
d,
x,
y,
x=x0;
y=y0;
d=2*dy-dx;
incrE=2*dy;
incrNE=2*(dy-dx);
while (x < x1 + 1)
{
SetPixel(x,y);
x++; / / next pixel
if (d<=0)
d+=incrE;
else
{
y++;
d+=incrNE;
}
}}
```

Questions

1. What is raster graphics? differentiate b/w raster and vector graphics?
2. Explain how Bresenham's algorithm takes advantage of the connectivity of pixels in drawing straight lines on a raster output device.
3. Explain midpoint line algorithm? Write alogorithm in your own words

Notes:

LESSON 3
RASTER GRAPHICS, BASIC RASTER GRAPHICS ALGORITHM, SCAN CONVERSION ALGO, OF LINE, CIRCLE ELLIPSE (CONTD…)

Today's Topics

- **Scan conversion algorithm of circle**
- **Scan conversion algorithm of ellipse**

Scan Converting Circles

For a circle $x^2 + y^2 = R^2$, we could use the following algorithm:

```
for (x=—R; x<=R; x++)
{
WritePixel(x, round(sqrt(R* R - x * x));
WritePixel(x, -round(sqrt(R* R - x * x));
}
```

However, this simple method is inefficient; further, the circle will have large gaps for the value of x close to R .

The large gaps can be avoided by using the parametric equations of the circle:

$$x = R\cos\Phi$$
$$y = R\sin\Phi$$

Also we can use 8 of symmetries of the circle: if we write a pixel (x , y), then we can also write

$$(x, -y), (-x, y), (-x, -y), (y, x), (-y, x), (-y, -x)$$

However the inefficiency problems remain; thus we look for an algorithm such as the midpoint algorithm.

Midpoint Circle Algorithm

```
void MidpointCircle( int radius, int value)
{
int x = 0, y = radius;
float d;
d = 5.0/4.0 -radius;
CirclePoints(x,y,value);

while (y > x)
{
if (d < 0) //inside circle
{
d += x * 2.0 + 3;
x++;
}
else //outside or on circle
{
d += (x-y)*2 + 5;
x++;
y—;
}
CirclePoints(x,y,value);
}}
```

Scan Converting Ellipses

Consider the standard ellipse of Fig. 3.19, centered at (0, 0). It is described by the equation

$F(x, y) = l/x^* + aV - a^2b^l = 0,$

where $2a$ is the length of the major axis along the x axis, and $2b$ is the length of the minor axis along the y axis. The midpoint technique discussed for lines and circles can also be applied to the more general conies. In this chapter, we consider the standard ellipse that is supported by SRGP;. Again, to simplify the algorithm, we draw only the arc of the ellipse that lies in the first quadrant, since the other three quadrants can be drawn by symmetry. Note also that standard ellipses centered at integer points other than the origin can be drawn using a simple translation. The algorithm presented here is based on Da Silva's algorithm, which combines the techniques used by Pitteway [PITT67], Van Aken [VANA84] and Kappel [KAPP85] with the use of partial differences [DASI89].

We first divide the quadrant into two regions; the boundary between the two regions is the point at which the curve has a slope of - 1

Determining this point is more complex than it was for circles, however. The vector that is perpendicular to the tangent to the curve at point P is called the *gradient,* defined as

grad $F(x, y) = dF/dx$ i + dF/dy j = $2tfx$ i + $2a^ly$ j.

The boundary between the two regions is the point at which the slope of the curve is - 1, and that point occurs when the gradient vector has a slope of 1-that is, when the i and j components of the gradient are of equal magnitude. The j component of the gradient is larger than the i component in region 1, and vice versa in region 2.

Thus, if at the next midpoint, $a^2(y_p - \frac{1}{2}) < b^2(x_p + 1)$, we switch from region 1 to region 2.

As with any midpoint algorithm, we evaluate the function at the midpoint between two pixels and use the sign to determine whether the midpoint lies inside or outside the ellipse and, hence, which pixel lies closer to the ellipse. Therefore, in region 1, if the current pixel is located at (x_p, y_p), then the decision variable for region 1, d_t, is $F(x, y)$ evaluated at $(x_p + 1, y_p - \frac{1}{2})$ the midpoint between E and SE. We now repeat the process we used for deriving the Fig. two Δs for the circle. For a move to E, the next midpoint is one increment over in x. Then,

$$d_M = F(x_p + 1, y_p - \frac{1}{2}) = b^2(x_p + 1)^2 + a^2(y_p - \frac{1}{2})^2 - a^2b^2,$$

$$d_{new}, = F(x_p + 2y_p - \frac{1}{2}) = (x_p + 2)^2 + a^2(y_p - \frac{1}{2})^2 - a^2b$$

Since $d_{new} = d_{old} + b^2(2x_p + 3)$, the increment $\Delta_E = b^2(2x_p + 3)$.

For a move to *SE*, the next midpoint is one increment over in *x* and one increment down in *y*. Then,

Since $d_{new} = d_{old} + b^2(2x_p + 3) + o^2(-2» + 2)$, the increment

$\Delta_{SB} = b^2\{2x_p + 3) + a^2(-2» + 2)$.

In region 2, if the current pixel is at $(x_p, »)$, the decision variable d_t is $F(x_p + \frac{1}{2}, y_p — 1)$, the midpoint between *S* and *SE*.

Computations similar to those given for region 1 may be done for region 2.

We must also compute the initial condition. Assuming integer values *a* and *b,* the ellipse starts at (0, *b),* and the first midpoint to be calculated is at (1, *b* - -£) Then,

$F(1, b - \frac{1}{2}) = b^2 + a^2 – (b - \frac{1}{2})^2 – ab^2 = b^2 + a^2(-b + \frac{1}{4})$.

At every iteration in region 1, we must not only test the decision variable d_1 and update the A functions, but also see whether we should switch regions by evaluating the gradient at the mid-point between *E* and *SE*. When the midpoint crosses over into region 2, we change our choice of the 2 pixels to compare from *E* and *SE* to *SE* and *S.* At the same time, we have to initialize the decision variable d_2 for region 2 to the midpoint between *SE* and *S.* That is, if the last pixel chosen in region 1 is located at (x_p, y_p), then the decision variable d_2 is initialized at

$(x_p + \backslash, y_p - 1)$. We stop drawing pixels in region 2 when the *y* value of the pixel is equal to 0.

As with the circle algorithm, we can either calculate the A functions directly in each iteration of the loop or compute them with differences. Da Silva shows that computation of second-order partials done for the as can, in fact, be used for the gradient as well. He also treats general ellipses that have been rotated and the many tricky90 Basic Raster Graphics Algorithms for Drawing 2D Primitives boundary conditions for very thin ellipses. The pseudocode algorithm uses the simpler direct evaluation rather than the more efficient formulation using second-ordeal differences; it also skips various tests. In the case of integer *a* and *b,* we can eliminate the fractions via program transformations and use only integer arithmetic.

Pseudocode for midpoint ellipse scan-conversion algorithm.

```
void Midpoint Ellipse (int a, int b, int value)
I* Assumes center of ellipse is at the origin. Note that overflow may occur */
/* for 16-bit integers because of the squares. */
{
double d2;
int x = 0;
int y = b;
double dl = b² - (a²b) + (0.25 a²);
EllipsePoints (x, v, value); I* The 4-way symmetrical WritePixel */
/* Test gradient if still in region 1 */
while ( a²{y - 0.5) > b²(x + 1) ) {      /* Region 1 */
if {dl < 0)                              /* Select E */
dl +=b²(2x + 3);
else {                                   /* Select SE */
dl += b²{2x + 3) + a²{-2y + 2);
y- - -
EllipsePoints {x, y, value);
 }   I* Region 1 */
d2 = b²{x + 0.5)² + a²{y - I)² - a²b²;
while (y> 0) {                           /* Region 2 */
if {d2 < 0) {                            /* Select SE */
d2 += b²{2x + 2) + a²{~2y + 3);
} else
d2+=a²(-2y + 3);                         /* Select S */
y- -\
EllipsePoints (x, y, value);
 }  I* Region 2 */ }
 /* MidpointEllipse */
```

Summary

Let's recall what we have covered in the lecture 2 and 3? We have seen a simple definition of Raster graphics . we have seen the difference b/w raster and vector ,basic algorithms of drawing 2d primitives , scan conversion algorithm of line, circle and ellipse ,some examples to demonstrate all these terminology. However, in the next lesson, we will work on graphics hardware

Questions

1. Explain algorithms for discovering if a point is inside or outside a circle?
2. Explain scan conversion algorithm of ellipse ? write algorithm also?
3. Explain scan conversion algorithm of circle ? write algorithm also?
4. How do generate a circle through three points?
5. How can the smallest circle enclosing a set of points be found?

Notes:

LESSON 4
GRAPHICS DISPLAY DEVICES, HARD COPY TECHNOLOGIES, DISPLAY TECHNOLOGIES

Topics Covered in the Unit

- **Basics of graphics hardware and software**
- **Graphics display devices**
- **Hard copy technologies**
- **Display technologies**
- **Raster and random scan display systems**
- **Video controller**

Today's Topics

- Basics of graphics hardware and software
- Graphics display devices
- Hard copy technologies
- Display technologies

The Basics of Graphics Hardware and Software

Hardware

"Vector graphics" Early graphic devices were line-oriented. For example, a "pen plotter" from H-P. Primitive operation is line drawing.

"Raster graphics" Today's standard. A raster is a 2-dimensional grid of pixels (picture elements). Each pixel may be addressed and illuminated independently. So the primitive operation is to draw a point; that is, assign a color to a pixel. Everything else is built upon that.

There are a variety of raster devices, both hardcopy and display.

Hardcopy:

Laser printer
Inkjet printer

Display

CRT (cathode ray tube)

LCD (liquid crystal display)

An important component is the "refresh buffer" or "frame buffer" which is a random-access memory containing one or more values per pixel, used to drive the display. The video controller translates the contents of the frame buffer into signals used by the CRT to illuminate the screen. It works as follows:

1. The display screen is coated with "phospors" which emit light when excited by an electron beam. (There are three types of phospor, emitting red, green, and blue light.) They are arranged in rows, with three phospor dots (R, G, and B) for each pixel.
2. The energy exciting the phosphors dissipates quickly, so the entire screen must be refreshed 60 times per second.
3. An electron gun scans the screen, line by line, mapping out a scan pattern. On each scan of the screen, each pixel is passed over once. Using the contents of the frame buffer, the controller controls the intensity of the beam hitting each pixel, producing a certain color.

Graphics Software

Graphics software (that is, the software tool needed to create graphics applications) has taken the form of subprogram libraries. The libraries contain functions to do things like:

draw points, lines, polygons
apply transformations
fill areas with color
handle user interactions

An important goal has been the development of standard hardware-independent libraries.

CORE
GKS (Graphical Kernel Standard)
PHIGS (Programmer's Hierarchical Interactive Graphics System)
X Windows
OpenGL

Hardware vendors may implement some of the OpenGL primitives in hardware for speed.

OpenGL:

gl: basic graphics operations
glu: utility package containing some higher-level modeling capabilities (curves, splines)
glut: toolkit. adds platform-independent functions for window management, mouse and keyboard interaction, pull-down menus
glui: adds support for GUI tools like buttons, sliders, etc.

Open Inventor. An object-oriented API built on top of OpenGL.

VRML. Virtual Reality Modeling Language. Allows creation of a model which can then be rendered by a browser plug-in.

Java3d. Has hierarchical modeling features similar to VRML.

POVray. A ray-tracing renderer

Hardcopy Devices

Categories

- Vector
  - Plotters
  - Raster
    - dot matrix printer
    - laser printer
    - inkjet printer

Characteristics

- dot size
- addressability (dots per inch)
  - inkjet: 2400x1200 dpi
  - laser: 1200x1200 dpi
- interdot distance = 1 / addressability
  - (Normally, dots overlap, so dot size > interdot distance)
- resolution = maximum number of distinguishable lines per inch. Depends on dot size, interdot distance, intensity distribution.

Displays

Most common: CRT (Cathode-ray tube) and LCD (Liquid crystal display)

CRT

- Electron gun sends beam aimed (deflected) at a particular point on the screen. Traces out a path on the screen, hitting each pixel once per cycle. "scan lines"
- Phosphors emit light (*phosphoresence*); output decays rapidly (exponentially - 10 to 60 microseconds)
- As a result of this decay, the entire screen must be redrawn (refreshed) at least 60 times per second. This is called the *refresh rate* . If the refresh rate is too slow, we will see a noticeable flicker on the screen.
- CFF (Critical Fusion Frequency) is the minimum refresh rate needed to avoid flicker. Depends to some degree on the human observer. Also depends on the *persistence* of the phosphors; that is, how long it takes for their output to decay.
- The *horizontal scan rate* is defined as the number of scan lines traced out per second.
- The most common form of CRT is the *shadow-mask* CRT. Each pixel consists of a group of three phosphor dots (one each for red, green, and blue), arranged in a triangular form called a *triad* . The shadow mask is a layer with one hole per pixel. To excite one pixel, the electron gun (actually three guns, one for each of red, green, and blue) fires its electron stream through the hole in the mask to hit that pixel.
- The *dot pitch* is the distance between the centers of two triads. It is used to measure the resolution of the screen.

(Note: On a vector display, a scan is in the form of a list of lines to be drawn, so the time to refresh is dependent on the length of the display list.)

LCD

A liquid crystal display consists of 6 layers, arranged in the following order (back-to-front):

- A reflective layer which acts as a mirror
- A horizontal polarizer, which acts as a filter, allowing only the horizontal component of light to pass through
- A layer of horizontal grid wires used to address individual pixels
- The liquid crystal layer
- A layer of vertical grid wires used to address individual pixels
- A vertical polarizer, which acts as a filter, allowing only the vertical component of light to pass through

How it works

- The liquid crystal rotates the polarity of incoming light by 90 degrees.
- Ambient light is captured, vertically polarized, rotated to horizontal polarity by the liquid crystal layer, passes through the horizontal filter, is reflected by the reflective layer, and passes back through all the layers, giving an appearance of lightness.
- However, if the liquid crystal molecules are charged, they become aligned and no longer change the polarity of light passing through them. If this occurs, no light can pass through the horizontal filter, so the screen appears dark.
- The principle of the display is to apply this charge selectively to points in the liquid crystal layer, thus lighting or not lighting points on the screen.
- Crystals can be dyed to provide color.
- An LCD may be backlit, so as not to be dependent on ambient light.
- TFT (thin film transistor) is most popular LCD technology today.

Interfacing Between the CPU and the Display

A typical video interface card contains a display processor, a frame buffer, and a video controller.

The frame buffer is a random access memory containing some memory (at least one bit) for each pixel, indicating how the pixel is supposed to be illuminated. The *depth* of the frame buffer measures the number of bits per pixel. A video controller then reads from the frame buffer and sends control signals to the monitor, driving the scan and refresh process. The display processor processes software instructions to load the frame buffer with data.

(Note: In early PCs, there was no display processor. The frame buffer was part of the physical address space addressable by the CPU. The CPU was responsible for all display functions.)

Some typical examples of frame buffer structures:

1. For a simple monochrome monitor, just use one bit per pixel.
2. A *gray-scale* monitor displays only one color, but allows for a range of intensity levels at each pixel. A typical example would be to use 6-8 bits per pixel, giving 64-256 intensity levels.

   For a color monitor, we need a range of intensity levels for each of red, green, and blue. There are two ways to arrange this.
3. A color monitor may use a color lookup table (LUT). For example, we could have a LUT with 256 entries. Each entry contains a color represented by red, green, and blue values.

We then could use a frame buffer with depth of 8. For each pixel, the frame buffer contains an index into the LUT, thus choosing one of the 256 possible colors. This approach saves memory, but limits the number of colors visible at any one time.

4. A frame buffer with a depth of 24 has 8 bits for each color, thus 256 intensity levels for each color. 2^{24} colors may be displayed. Any pixel can have any color at any time. For a 1024x1024 monitor we would need 3 megabytes of memory for this type of frame buffer.

The display processor can handle some medium-level functions like scan conversion (drawing lines, filling polygons), not just turn pixels on and off. Other functions: bit block transfer, display list storage.

Use of the display processor reduces CPU involvement and bus traffic resulting in a faster processor.

Graphics processors have been increasing in power faster than CPUs, a new generation every 6-9 months. example: NVIDIA GeForce FX

- 125 million transistors (GeForce4: 63 million)
- 128MB RAM
- 128-bit floating point pipeline

One of the advantages of a hardware-independent API like OpenGL is that it can be used with a wide range of CPU-display combinations, from software-only to hardware-only. It also means that a fast video card may run slowly if it does not have a good implementation of OpenGL.

Input Devices

- Mouse or trackball
- Keyboard
- Joystick
- Tablet
- Touch screen
- Light pen

Questions

1. Explain about the display technologies?
2. Explain various display devices?
3. What are the different hardware and software of graphics?
4. List five graphic soft copy devices for each one briefly explain?
   A. How it works.
   B. Its advantages and limitations.
   C. The circumstances when it would be more useful.
5. List five graphic hard copy devices for each one briefly explain?
   D. How it works.
   E. Its advantages and limitations.
   F. The circumstances when it would be more useful.

Notes:

LESSON 5
RASTER AND RANDOM SCAN DISPLAY SYSTEMS

Today's Topicss

- Raster and random scan display systems

Raster-scan Display Systems

We discuss the various elements of a raster display, stressing two fundamental ways in which various raster systems differ one from another.

First, most raster displays have some specialized hardware to assist in scan converting output primitives into the pixmap, and to perform the raster operations of moving, copying, and modifying pixels or blocks of pixels. We call this hardware a graphics display processor. The fundamental difference among display systems is how much the display processor does versus how much must be done by the graphics subroutine package executing on the general-purpose CPU that drives the raster display. Note that the graphics display processor is also sometimes called a graphics controller (emphasizing its similarity to the control units for other peripheral devices) or a display coprocessor. The second key differentiator in raster systems is the relationship between the pixmap and the address space of the general-purpose computer's memory, whether the pixmap is part of the general-purpose computer's memory or is separate.

We introduce a simple raster display consisting of a CPU containing the pixmap as part of its memory, and a video controller driving a CRT. There is no display processor, so the CPU does both the application and graphics work.

Simple Raster Display System

The simplest and most common raster display system organization is shown in The relation between memory and the CPU is exactly the same as in a non graphics', computer system. However, a portion of the memory also serves as the pixmap. The video if controller displays the image defined in the frame buffer, accessing the memory through a separate access port as often as the raster-scan rate dictates. In many systems, a fixed portion of memory is permanently allocated to the frame buffer, whereas some systems have several interchangeable memory areas (sometimes called pages in the personal-computer world). Yet other systems can designate (via a register) any part of memory for the frame buffer. In this case, the system may be organized, or the entire system memory may be dual-ported.

The application program and graphics subroutine package share the system memory and are executed by the CPU. The graphics package includes scan-conversion procedures, so that when the application program calls, say, SRGP_ line Cord (xl, yl, x2, y2), the graphics package can set the appropriate pixels in the frame buffer . Because the frame buffer is in the address space of the CPU, the graphics package can easily access it to set pixels and to implement the Pix instructions.

The video controller cycles through the frame buffer, one scan line at a time, typically 60 times per second. Memory reference addresses are generated in synchrony with the raster scan, and the contents of the memory are used to control the CRT beam's intensity or

A simple raster display system architecture. Because the frame buffer may be stored anywhere in system memory, the video controller accesses the memory via the system bus. for the video controller is organized . The raster-scan generator, uses deflection signals that generate the raster scan; it also controls the X and Y address registers, which in turn define the memory location to be accessed next. : Assume that the frame buffer is addressed in x from 0 to and in y from 0 to y then, at the start of a refresh cycle, the X address register is set to zero and the Y register is set to (the top scan line). As the first scan line is generated, the X address is incremented up through each pixel value is fetched and is used to control the intensity of the CRT beam. After the first scan line, the X address is reset to zero and the Y address is decremented by one. The process continues until the last scan line (y = 0) is generated.

In this simplistic situation, one memory access is made to the frame buffer for each pixel to be displayed. For a medium-resolution display of 640 pixels by 480 lines refreshed 60 times per second, a simple way to estimate the time available for displaying a single 1-bit pixel is to calculate 1/(480 x 640 x 60) = 54 nanoseconds. This calculation ignores their fact that pixels are not being displayed during horizontal and vertical retrace . But typical RAM memory chips have cycle times around 200 nanoseconds: They*' cannot support one access each 54 nanoseconds! Thus, the video controller must fetch multiple pixel values in one memory cycle. In the case at hand, the controller might fetch 16 bits in one memory cycle, thereby taking care of 16 pixels X 54 ns/pixel = 864 nanoseconds of refresh time. The 16 bits are loaded into a register on the video controller, then are shifted out to control the CRT beam intensity, one each 54 nanoseconds. In the 864 nanoseconds this takes, there is time for about four memory cycles: one for the video controller and three for the CPU. This sharing may force the CPU to wait for memory accesses, potentially reducing the speed of the CPU by 25 percent. Of course, cache memory on the CPU chip can be used to ameliorate this problem.

It may not be possible to fetch 16 pixels in one memory cycle. Consider the situation when the pixmap is implemented with five 64-KB-memory chips, with each chip able to deliver 1 bit per cycle (this is called a 64-KB by 1 chip organization), for a total of 5 bits in the 200-nanoseconds cycle time. This is an average of 40 nanoseconds per bit (i.e., per pixel), which is not much faster than the 54 nanoseconds/pixel scan rate and leaves hardly any time for accesses to the memory by the CPU (except during the approximately 7-microsecond inter-scan-line retrace time and 1250-microsecond inter frame vertical retrace time).

With five 32-KB by 2 chips, however, 10 pixels are delivered in 200 nanoseconds, leaving slightly over half the time available for the CPU. With a 1600 by 1200 display, the pixel time is 1/(1600 x 1200 x 60) = 8.7 nanoseconds. With a 200-nanoseconds memory cycle time, 200/8.7 = 23 pixels must be fetched each cycle. A 1600 x 1200 display needs 1.92 MB of memory, which can be provided by eight 256-KB chips. Again, 256-KB by 1 chips can provide only 8 pixels per cycle: on the other hand, 32-KB by 8 chips can deliver 64 pixels, freeing two-thirds of the memory cycles for the CPU.

Access to memory by the CPU and video controller is clearly a problem: . The solution is RAM architectures that accommodate the needs of raster displays.

We have thus far assumed monochrome, 1-bit-per-pixel bitmaps. This assumption is fine for some applications, but is grossly unsatisfactory for others. Additional control over the intensity of each pixel is obtained by storing multiple bits for each pixel: 2 bits yield four intensities, and so on. The bits can be used to control not only intensity, but also color. How many bits per pixel are needed for a stored image to be perceived as having continuous shades of gray? Five or 6 bits are often enough, but 8 or more bits can be necessary. Thus, for color displays, a somewhat simplified argument suggests that three times as many bits are needed: 8 bits for each of the three additive primary colors red, blue, and green .

Systems with 24 bits per pixel are still relatively expensive, however, despite the decreasing cost of solid-state RAM. Furthermore, many color applications do not require 2^{24} different colors in a single picture (which typically has only 2^{18} to 2^{20} pixels). On the other hand, there is often need for both a small number of colors in a given picture .

Percentage of Time an Image is Being Traced Iring Which the Processor can Access the Memory Taining the Bitmap*
A 200-nanosecond memory cycle km and 60-Hz display rate are assumed throughout. The pixel time for a **1512** X 512 display is assumed to be 64 nanoseconds; that for 1024 x 1024, 16 nanoseconds. These times are **Liberal,** since they do not include the horizontal and vertical retrace times; the pixel times are actually about 45 .1 .11.5 nanoseconds, respectively.

Implication and the ability to change colors from picture to picture or from application to application. Also, in many image-analysis and image-enhancement applications, it is desirable to change the visual appearance of an image without changing the underlying data defining the image, in order, say, to display all pixels with values below some threshold as black, to expand an intensity range, or to create a pseudocolor display of a monochromatic image.

For these various reasons, the video controller of raster displays often includes a video look-up table (also called a look-up table or LUT). The look-up table has as many entries as there are pixel values. A pixel's value is used not to control the beam directly, but rather as an index into the look-up table. The table entry's value is used to control the intensity or color of the CRT. A pixel value of 67 would thus cause the contents of table entry 67 to be accessed and used to control the CRT beam. This look-up operation is done for each pixel on each display cycle, so the table must be accessible quickly, and the CPU must be able to load the look-up table on program command.

The look-up table is interposed between the frame buffer and the CRT display. The frame buffer has 8 bits per pixel, and the look-up table therefore has 256 entries.

The simple raster display system organizations used in many inexpensive personal computers. Such a system is inexpensive to build, but has a number of disadvantages. First, scan conversion in software is slow. For instance, the (x, y) address of each pixel on a line must be calculated, then must be translated into a memory address consisting of a byte and bit-within-byte pair. Although each of the individual steps is simple, each is repeated many times. Software-based scan conversion slows down the overall pace of user interaction with the application, potentially creating user dissatisfaction. The second disadvantage of this architecture is that as the addressability or the refit rate of the display increases, the number of memory accesses made by the video controller also increases, thus decreasing the number of memory cycles available to the CPU. CPU is thus slowed down, especially with the architecture. With dual porting of part of the system memory shown in the slowdown occurs when the CPU is accessing the frame buffer, usually for scan conversion or operations. These two disadvantages must be weighed against the ease with which the (can access the frame buffer and against the architectural simplicity of the system).

Questions

1. What is the difference between Raster and random scan display systems?
2. Explain Simple Raster Display System?

Notes:

LESSON 6
RASTER AND RANDOM SCAN DISPLAY SYSTEMS AND VIDEO CONTROLLER

Today's Topics

- **Raster Display System with Peripheral Display Processor**
- **Video controller**

Raster Display System with Peripheral Display Processor

The raster display system with a peripheral display processor is a common architecture that avoids the disadvantages of the simple raster display by introducing separate graphics processor to perform graphics functions such as scan conversion raster operations, and a separate frame buffer for image refresh. We now have two processors: the general-purpose CPU and the special-purpose display processor. We also! have three memory areas: the system memory, the display-processor memory, and thy frame buffer. The system memory holds data plus those programs that execute on the CPU the application program, graphics package, and operating system. Similarly, the display-] processor memory holds data plus the programs that perform scan conversion and j operations. The frame buffer contains the displayable image created by the scan-con verse and raster operations.

In simple cases, the display processor can consist of specialized logic to perform I mapping from 2D *(x, y)* coordinates to a linear memory address. In this case, scan-conversion and raster operations are still performed by the CPU, so the display processor memory is not needed; only the frame buffer is present. Most peripheral display! processors also perform scan conversion. In this section, we present a prototype system. Its features are a (sometimes simplified) composite of many typical commercially available systems, such as the plug-in graphics cards used with IBM's PC, XT, AT, PS, and compatible computers.

The frame buffer is 1024 by 1024 by 8 bits per pixel, and there is a 256-entry look-up table of 12 bits, 4 each for red, green, and blue. The origin is at lower left, but only the first 768 rows of the pixmap (y in the range of 0 to 767) are displayed. The display has six status registers, which are set by various instructions and affect the execution of other instructions. These are the CP (made up of the X and Y position registers), FILL, INDEX, WMODE, MASK, and PATTERN registers. Their operation is explained next. The instructions for the simple raster display are as follows:

Move *(x, y)* The X and Y registers that define the current position (CP) are set to *x* and y. Because the pixmap is 1024 by 1024, *x* and *y* must be between 0 and 1023. MoveR *(dx, dy)* the values *dx* and *dy* are added to the X and Y registers, thus defining a new CP. The *dx* and *dy* values must be between -1024 and +1023, and are represented in 2 complement notation. The addition may cause overflow and hence a wraparound of the X and Y register values. Line*(x, y)* A line is drawn from CP to *(x, y)*, and this position becomes the new CP. LineR *(dx, dy)* A line is drawn from CP to CP + *(dx, dy)*, and this position becomes the new CP. Point *(x, y)* The pixel at *(x, y)* is set, and this position becomes the new CP. PointR *(dx, dy)* The pixel at CP + *(dx, dy)* is set, and this position becomes the new CP.

A single-address-space (SAS) raster display system architecture with an integral display processor. The display processor may have a private memory for algorithms and working storage results of scan conversion can go either into the frame buffer for immediate display, or elsewhere in system memory for later display. Similarly, the source and destination for raster operations performed by the display processor can be anywhere in system memory (now the only memory of interest to us). This arrangement is also attractive because the CPU can directly manipulate pixels in the frame buffer simply by reading or writing the appropriate bits.

SAS architecture has, however, a number of shortcomings. Contention for access to the system memory is the most serious. We can solve this problem at least partially by dedicating a special portion of system memory to be the frame buffer and by providing a second access port to the frame buffer from the video controller, as shown in Fig. 4.24. Another solution is to use a CPU chip containing instruction- or data-cache memories, thus reducing the CPU's dependence on frequent and rapid access to the system memory. Of course, these and other solutions can be integrated in various ingenious ways, as discussed in more detail . In the limit, the hardware PixBlt may work on only the frame buffer. What the application programmer sees as a single Pix Blt instruction may be treated as several different cases, with software simulation if the source and destination are not supported by the hardware. Some processors are actually fast enough to do this, especially if they have an instruction-cache memory in which the tight inner loop of the software simulation can remain.

As suggested earlier, nontraditional memory-chip organizations for frame buffers alsc can help to avoid the memory-contention problem. One approach is to turn on all the pixel-on a scan line in one access time, thus reducing the number of memory cycles needed to scan convert into memory, especially for filled areas. The video RAM (VRAM) organization, developed by Texas Instruments, can read out all the pixels on a scan line in one cycle, thus reducing the number of memory cycles needed to refresh the display. Again.

A more common single-address-space raster display system architecture with an integral display processor. The display processor may have a private memory for algorithms and working storage. A dedicated portion of the system memory is dual-ported so that it can be accessed directly by the video controller, without the system bus being tied up.

Another design complication arises if the CPU has a virtual address space, as do the commonly used Motorola 680x0 and Intel 80x86 families, and various reduced-instruction-set-computer (RISC) processors. In this case memory addresses generated by the display processor must go through the same dynamic address translation as other memory addresses. In addition, many CPU architectures distinguish between a kernel operating system virtual address space and an application program virtual address space. It is often desirable for the frame buffer (canvas 0 in SRGP terminology) to be in the kernel space, so that the operating system's display device driver can access it directly.

However, the canvases allocated by the application program must be in the application space. Therefore display instructions which access the frame buffer must distinguish between the kernel and application address spaces. If the kernel is to be accessed, then the display instruction must be invoked by a time-consuming operating system service call rather than by a simple subroutine call.

Despite these potential complications, more and more raster display systems do in fact have a single-address-space architecture, typically of the type in Fig. 4.24. The flexibility of allowing both the CPU and display processor to access any part of memory in a uniform and homogeneous way is very compelling, and does simplify programming.

The Video Controller

The most important task for the video controller is the constant refresh of the display. There are two fundamental types of refresh: *interlaced* and *noninterlaced*. The former is used in broadcast television and in raster displays designed to drive regular televisions. The refresh cycle is broken into two fields, each lasting-gg-second; thus, a full refresh lasts-gj second. All odd-numbered scan lines are displayed in the first field, and all even-numbered ones ; displayed in the second. The purpose of the interlaced scan is to place some ne information in all areas of the screen at a 60-Hz rate, since a 30-Hz refresh rate tends to cause flicker. The net effect of interlacing is to produce a picture whose effective refresh r is closer to 60 than to 30 Hz. This technique works as long as adjacent scan lines do in display similar information; an image consisting of horizontal lines on alternating scan lin would flicker badly. Most video controllers refresh at 60 or more Hz and use noninterlaced scan.

The output from the video controller has one of three forms: RGB, monochrome, i NTSC. For RGB (red, green, blue), separate cables carry the red, green, and blue signals to control the three electron guns of a shadow-mask CRT, and another cable carries the synchronization to signal the start of vertical and horizontal retrace. There are standards for the voltages, wave shapes, and synchronization timings of RGB signals. For 480-scan-linej monochrome signals, RS-170 is the standard; for color, RS-170A; for higher-resolutionl monochrome signals, RS-343. Frequently, the synchronization timings are included on the s same cable as the green signal, in which case the signals are called *composite video.* Monochrome signals use the same standards but have only intensity and synchronization' cables, or merely a single cable carrying composite intensity and synchronization.

NTSC (National Television System Committee) video is the signal format used in North American commercial television. Color, intensity, and synchronization information is combined into a signal with a bandwidth of about 5 MHz, broadcast as 525 scan lines, in two fields of 262.5 lines each. Just 480 lines are visible; the rest occur during the vertical retrace periods at the end of each field. A monochrome television set uses the intensity and synchronization information; a color television set also uses the color information to control the three color guns. The bandwidth limit allows many different television channels to broadcast over the frequency range allocated to television. Unfortunately, this bandwidth limits picture quality to an effective resolution of about 350 by 350. Nevertheless, NTSC is the standard for videotape-recording equipment. Matters may improve, however, with increasing interest in 1000-line high-definition television (HDTV) for videotaping and satellite broadcasting. European and Soviet television broadcast and videotape standards are two 625-scan-line, 50-Hz standards, SECAM and PAL.

Some video controllers superimpose a programmable cursor, stored in a 16 by 16 or 32 by 32 pixmap, on top of the frame buffer. This avoids the need to PixBlt the cursor shape into the frame buffer each refresh cycle, slightly reducing CPU overhead. Similarly, some video controllers superimpose multiple small, fixed-size pixmaps (called *sprites)* on top of the frame buffer. This feature is used often in video games.

Questions

1. What is video controller?
2. What is raster and random scan display systems?

Notes:

LESSON 7
CLIPPING IN A RASTER WORLD, CLIPPING LINES

Topics Covered in the Unit

- **Clipping**
- **Clipping in a Raster world**
- **Clipping lines**
- **Cohen - Sutherland algorithm**
- **Cyrus beck algorithm.**

Learning Objectives

Upon completion of this unit, the student will be able to :

- Explain Clipping
- Explain clipping in a raster world
- Demonstrate an understanding of the elementary algorithm for line Clipping
- Demonstrate an understanding of the cohen-sutherland algorithm of Clipping
- Demonstrate an understanding of the Cyrus beck algorithm for Clipping

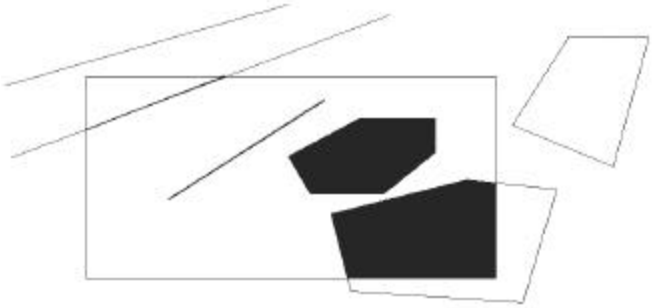
Today's Topics

- Clipping
- Clipping in a Raster world

Clipping

Introduction

Clipping a geometric object A to a clipping polygon W is the process of removing those parts of A which are not contained in W. The following illustration shows several lines and polygons being clipped to a rectangular window:

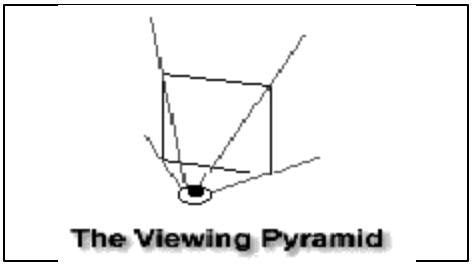


We will look at algorithms for clipping lines and polygons.

Question: Should we clip before, after, or during scan conversion?

What exactly is clipping and why is it needed? Clipping in 3D is just like the real-life clipping that you think of - taking a polygon and "snipping" off a part of it. Why do we need to do this? Well, often we don't need parts of a polygon to be displayed or for the vertices to be transformed. We have already seen one case where this happens: we don't want the polygon to reach "in front of" the screen... or "behind the camera," whichever one you prefer. In short, we should clip all polygons with vertices whose $z<0$, so they become polygons with vertices all having $z<0$ and $z=0$.

Before we try clipping in 3D, however, let's consider another case where we'd want our polygons to be clipped. Namely: we don't want our polygons being drawn somewhere off the screen. In fact, we can eliminate all the polygons, which are beyond the limits of the screen, before we do any Cam2Screen translation, and we shouldn't actually attempt to draw them. So we should clip polygons to the four sides of the screen. What would this mean in 3D? Well, imagine a "viewing pyramid" with its tip at the eye, which encompasses all the viewable area:



Go here to learn about planes and vectors, if you aren't very familiar with them.

This viewing pyramid consists of four planes: left, right, top and bottom. Add to that the $z=0$ plane, and we have the main clipping frustum. This frustrum will affect the entire scene. A frustrum is basically a 3D clipping area.

Why use 3D clipping, rather than 2D clipping? Because 3D clipping can accomplish everything 2D clipping can, plus more. Clipping to a 2D line is like clipping to a 3D plane, but the plane can be oriented many more different ways in 3D. Also, 3D clipping is inherently perspective-correct, since it's all done in 3D. Finally, when polygons are clipped in 3D, the resulting polygons can be directly rendered - that is, texture mapped by any texture mapped. The coordinates are already there

Clipping in a Raster World

It is essential that both clipping and scan conversion be done as rapidly as possible, in order to provide the user with quick updated resulting from changes to the application model. Clipping can be done analytically, on the fly during scan conversion, or as part of a copy Pixel with the desired clip rectangle from a canvas storing unclipped primitives to the destination canvas. Combining clipping and scan conversion, sometimes called *scissoring*, is easy to do for filled or thick primitives as part of span arithmetic: Only the extreme need to be clipped, and no interior pixels need be examined. Scissoring shows yet another advantage of span coherence. Also, if an outline primitive is not much larger than the clip rectangle, not many pixels, relatively speaking, will fall outside the clip region. For such a case, it may well be faster to generate each pixel and to clip it (i.e., to write it conditionally) then to do analytical clipping beforehand. In particular, although the bounds test is

in the inner loop, the expensive memory write is avoided for exterior pixels, and both the incremental computation and the testing may run entirely in a fast memory, such as a CPU instruction catch or a display controller's micro code memory.

Other tricks may be useful. For example, one may "home in" on the intersection of a line with a clip edge by doing the standard midpoint scan-conversion algorithm on every *i*th pixel and testing the chosen pixel against the rectangle bounds until the first pixel that lies inside the region is encountered. Then the algorithm has to back up, find the first pixel inside, and to do the normal scan conversion thereafter. The last interior pixel could be similarly determined, or each pixel could be tested as part of the scan-conversion loop and scan conversion stopped the first time the test failed. Testing every eighth pixels to back up.

For graphics packages that operate in floating point, it is best to clip analytically in the floating-point coordinate system and then to scan convert the clipped primitives, being careful to initialize decision variables correctly, For integer graphics packages such as SRGP, there is a choice between preclipping and then scan converting or doing clipping during scan conversion. Since it is relatively easy to do analytical clipping for lines and polygons, clipping of those primitives is often done before scan conversion, while it is faster to clip other primitives during scan conversion. Also, it is quite common for a floating – point graphics package to do analytical clipping in its coordinate system and then to can lower-level scan-conversion software that actually generates the clipped primitives; this integer graphics software cold then do an additional raster clip to rectangular (or even arbitrary) window boundaries. Because analytic clipping of primitives is both useful for integer graphics packages and essential for 2D and #D floating point graphics packages, we discuss the basic analytical clipping algorithms in this unit in next lecture

Questions

1. What is clipping?
2. Explain clipping in a raster world?

Notes:

LESSON 8
CLIPPING LINES

Today's Topics

- **Clipping Lines**

Clipping Lines

This section treats analytical clipping of lines against rectangles; algorithms for clipping other primitives are handled in subsequent sections. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that SRGP primitives built out of lines 9 i.e., ploylines, unfilled rectangles, and polygons) can be clipped by repeated application of the line clipper. Furthermore, circles and ellipses may be piecewise linearly approximated with a sequence of very short lines, so that boundaries can be treated as a single polyline or polygon for both clipping and scan conversion. Conics are represented in some systems as ratios of parametric polynomials), a representation that also lends itself readily to an incremental, piecewise linear approximation suitable for a line-clipping algorithm. Clipping a rectangle against a rectangle results in at most a single rectangle. Clipping a convex polygon against a rectangle results in at most a single convex polygon, but clipping a concave polygon against a rectangle results in at most a single convex polygon, but clipping a concave polygon may produce more than one concave polygon. Clipping a circle or ellipse against a rectangle results in as many as four arcs.

Lines intersecting a rectangular clip region (or any convex polygon) are always clipped to a single line segment; lines lying on the clip rectangle's border are considered inside and hence are displayed.

Clipping Lines by Solving Simultaneous Equations

To clip a line, we need to consider only its endpoints, not its infinitely many interior points. If both endpoints of a line lie inside the clip rectangle), the entire line lies inside the clip rectangle and can be *trivially accepted.* If one endpoint lies inside and one outside (e.g., *CD* in the figure), the line intersects the clip rectangle and we must compute the intersect with the clip rectangle (*EF, GH AND IJ* in the figure), and we need to perform further calculations to determine whether there are any intersections, and if there are, where they occur.

The brute-force approach to clipping a line that cannot be trivially accepted is to intersect that line with each of the four clip-rectangle edges to see whether any intersection points lie on those edges; if so, the line cuts the clip rectangle and is partially inside. For each line and clip - rectangle edge, we therefore take the two mathematically infinite lines that contain them and intersect them. Next, we test whether this intersection point is "interior"- that is, whether it lies within both the clip rectangle edge and the line; if so, there is an intersection with the clip rectangle, intersection points *G'* and *H* are interior, but *I'* and *J'* are not.

When we use this approach, we must solve two simultaneous equations using multiplication and division for each <edge, line> pair. Although the slop-intercept formula for lines learned in analytic geometry could be used, it describes infinite lines, whereas in graphics and clipping we deal with finite lines (called line segments in mathematics). In addition, the slop-intercept formula does not deal with vertical lines-a serious problem, given our upright clip rectangle. A parametric formulation for line segments solves both problems:

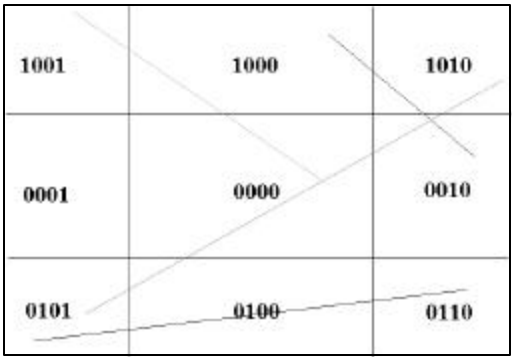
$X=x_0+t(x_1-x_0), y_0+t(y_1-y_0).$

These equations describe (x,y) on the directed line segment from (x_0, y_0) to (x_1, y_1) for the parameter t in the range [o, 1], as simple substitution for t confirms. Two sets of simultaneous equations of this parametric form can be solved for parameters t_{edge} for the edge and t_{line} for the line segment. The values of t_{edge} and t_{line} can then be checked to see whether both lie in [0, 1]; if they do, the intersection point lies within both segments and is a true clip-rectangle intersection. Furthermore, the special case of a line parallel to a clip-rectangle edge must also be tsted before the simultaneous equations can be solved. Altogether, the brute-force approach involves considerable calculation and testing ; it is thus inefficient.

Clipping Lines

There are two well-known approaches to line clipping algorithms: the Cohen-Sutherland algorithm and the parametric approach used in the Cyrus-Beck and Liang-Barsky algorithms. Cohen-Sutherland and Liang-Barsky are specifically formulated to clip to an upright rectangle; Cyrus-Beck can be used to clip to any convex polygon. First, we consider the Cohen-Sutherland algorithm.

Cohen-Sutherland algorithm view the plane as being divided by the edges of the clipping rectangle into 9 regions, like a tic-tac-toe board. Each region is assigned an "out code": a 4-bit string which indicates the "out ness" of the region with respect to the 4 clipping edges:



Question

For a given line, the algorithm does the following:

1. Determine the out codes the endpoints of the line.
2. If the OR operation applied to the out codes is zero, the line lies entirely within the clip rectangle and can be trivially accepted.
3. If the AND operation applied to the out codes is nonzero, the entire line lies outside one of the clipping edges, so the line can be trivially rejected.
4. For the nontrivial case:
   a. Select an endpoint with a nonzero out code, and a clip rectangle edge, which is outside (that is, for which the out code contains a '1').
   b. Determine the point of intersection of the line with the edge.
   c Replace the selected endpoint with the point of intersection found in step b.

Compute the out code of the new endpoint.

5. Repeat beginning with step 2, until the line can be trivially accepted or rejected.

Notes:

LESSON 9
COHEN – SUTHERLAND ALGORITHM

Today's Topics

- **Cohen – Sutherland Algorithm**

The algorithm of sutherland-hodgeman

The Cohen-Sutherland Line-Clipping Algorithm

The more efficient Cohen-Sutherland algorithm performs initial tests on a line to determine whether intersection calculations can be avoided. First, endpoint pairs are checked for trivial acceptance. If the line cannot be trivially accepted, *region checks* are done. For instance, two simple comparisons on x show that both endpoints of line *EF* in Fig. 3.38 have an x coordinate less than x_{min} and thus lie in the region to the left of the clip rectangle (i.e., in the outside half plane defined by the left edge); therefore, line segment *EF* can be *trivially rejected* and needs to be neither clipped nor displayed. Similarly, we can trivially reject lines with both endpoints in regions to the right of X_{max}, below Y_{min}, and above $Y_{max.}$

If the line segment can be neither trivially accepted nor rejected, it is divided into two segments at a clip edge, so that one segment can be trivially rejected. Thus, a segment is iteratively clipped by testing for trivial acceptance or rejection, and is then subdivided if neither test is successful, until what remains is completely inside the clip rectangle or can be trivially rejected. The algorithm is particularly efficient for two common cases. In the first case of a large clip rectangle enclosing all or most of the display area, most primitives can be trivially accepted. In the second case of a small clip rectangle, almost all primitives can be trivially rejected. This latter case arises in a standard method of doing pick correlation in which a small rectangle surrounding the cursor, called the *pick window,* is used to clip primitives to determine which primitives lie within a small (rectangular) neighborhood of the cursor's pick point

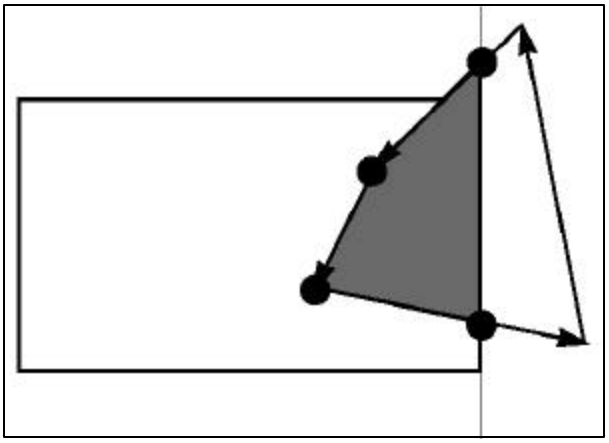
To perform trivial accept and reject tests, we extend the edges of the clip rectangle to divide the plane of the clip rectangle into nine regions). Each region is assigned a 4 - bit code, determined by where the region lies with respect to the outside half planes of the clip- rectangle edges. Each bit is the out code is set to either 1 (true) or 0 (false); the 4 bits in the code correspond to the following conditions:

The algorithm to perform sutherland-hodgeman clipping is very simple and easy to implement. Grab a piece of paper, and draw a convex polygon on it. Then draw a rectangular 'screen' area, so that parts of the polygon are outside this area. Number the vertices of your polygon, 0...n. Now start with the first edge. The startpoint of this edge is called 'v1', the endpoint of the edge is 'v2'. Make a table, called 'cv'. Now use the following algorithm:

1. If v1 is 'in' and v2 is 'out', the edge is apparently 'going out'. Determine the point where the edge leaves the 'screen'. Call this point 'clipped coord'. Note this in your 'cv' table.
2. If v1 is 'out' and v2 is 'in', the edge is apparently 'coming in'. Determine the point of entrance. Write this point in your 'cv' table. Also write 'v2' in your 'cv' table. Lines that are entering the area always cause two entries in the 'cv' table.
3. If both v1 and v2 are 'in', write only v2 to your 'cv' table.
4. If neither v1 nor v2 are 'in', don't do anything.

Note that if each of these cases would have occurred, exactly four vertices where written in 'cv'.

When you have done this for your first edge, rename 'v2' to 'v1', 'dist2' to 'dist1' and so on, and get 'v2', 'dist2' for the second edge. Then choose one of the four steps mentioned above again. When all four edges are processed, your 'cv' table contains the clipped polygon



Parametric Line Clipping

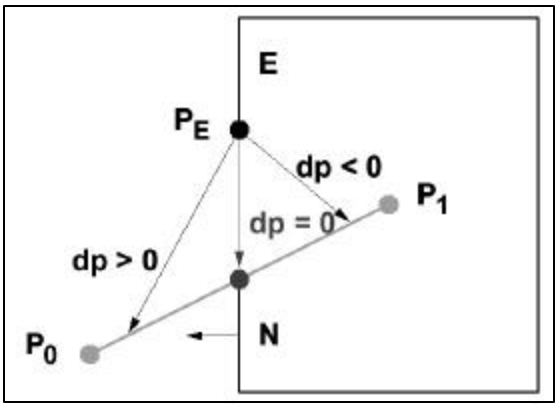
Background: The equation for a line containing points P_0 and P_1 can be written:

$P = P_0 + t * (P_1 - P_0)$

where $(P_1 - P_0)$ represents the vector from P_0 to P_1.

When t is between 0 and 1, the corresponding point P lies on the line segment between P_0 and P_1. That makes this form of the equation particularly useful, because we are generally interested in line segments defined between two endpoints. When we are looking for, say, the intersection of two line segments, we can compute the t values at the intersection point. If either t value is less than zero or greater than one, we know that the two segments do not intersect.

We can use the concept of a normal vector to determine the point of intersection between two lines. Normal's can also be used to determine whether a point lies on the inside or outside of an edge. Given a line segment from $P_0(x_0,y_0)$ to $P_1(x_1,y_1)$, we can compute a rightward-pointing vector normal to P_0P_1 as $N(y_1-y_0,-(x_1-x_0))$.



Suppose we want to find the point of intersection between the line P_0P_1 and a second line whose normal is N. Choose an arbitrary point P_e on the second line. Then for any point P, compute the dot product of $(P-P_e)$ and N. If the dot product is > 0, P is on the same side of the line as the direction of the normal, if the dot product is < 0, P is on the opposite side, and if the dot product is 0, P is on the line. This gives us a method for determining the intersection point of two lines: A point $P = P_0 + t * (P_1 - P_0)$ is an intersection point if $(P - P_e) \bullet N$ is 0:

$(P_0 + t * (P_1 - P_0) - P_e) \bullet N = 0$

That is,

$(P_0 - P_e) \bullet N + t * (P_1 - P_0) \bullet N = 0$

Solving for t gives:

$t = - (P_0 - P_e) \bullet N / (P_1 - P_0) \bullet N$

Questions

1. Explain The Cohen-Sutherland Line-Clipping Algorithm?

Notes:

LESSON 10
CYRUS BECK ALGORITHM

Today's Topics

- **Cyrus Beck Algorithm.**

Cyrus-Beck Techniques (1978): A Parametric Line-Clipping Algorithm

Cohen-Sutherland algorithm can only trivially accept or reject lines within the given bounding volume; it cannot calculate the exact intersection point. But, the parametric line clipping algorithm can calculate the value of the parameter t , where the two lines intersect. This can be easily understood by looking at the following picture and pseudo code:

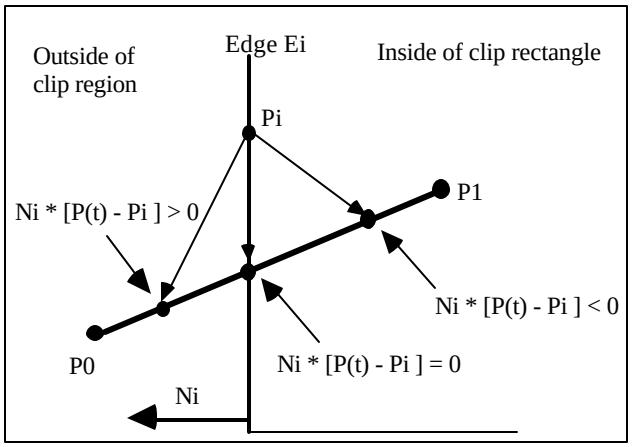


Figure 1: Dot Product for 3 points outside, inside, and on the boundary of the clip region.

The line is parametrically represented by P(t) = P0 + t (P1 - P0)

% Pseudo Code for Cyrus Beck Parametric Line-Clipping Algorithm

```
{
precalculate Ni and select a Pi for each edge Ei
for (each line segment to be clipped) {
            if (P1 = P0)
        line is degenerate so clip as a point;
            else {
                    D = P1 - P0;
                    te = 0;
                    tl  = 1;
        for (each candidate intersection with a clip edge) {
         if (Ni * D # 0)  {
                                t =  - { Ni * [P0 - Pi] }  /
(Ni * D)
                                if (Ni * D > 0)
                                    tl = min (tl, t);
                                else
                                    te = max (te, t);
                                }
                            }

                            if (te > tl)
                               return nil;
                            else
                               return P(te) and P(tl) as true clip
intersection points;
                        }
                }
}
```

The Cyrus-Beck Algorithm

The basic idea of the algorithm (Cyrus-Beck) is as follows:

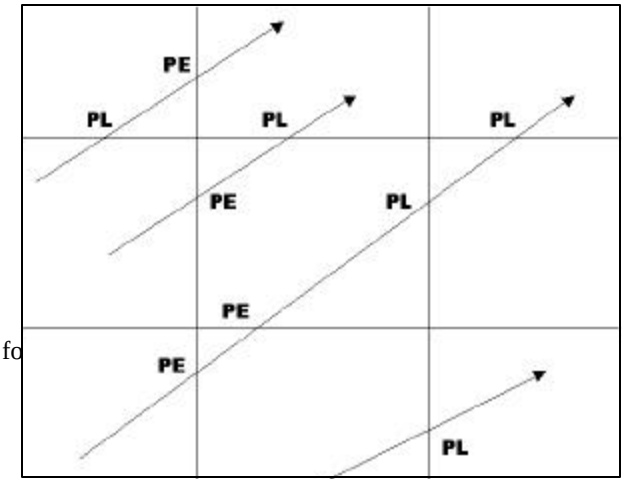
The line to be clipped is expressed using its parametric representation. For each edge of the clipping polygon, we are given a point P_e on the edge, and an outward-pointing normal N. (The vertices of the clipping polygon are traversed in the counter-clockwise direction.) The objective is to find the t values where the line enters and leaves the polygon (t_E and t_L), or to determine that the line lies entirely outside the polygon.

t_E is initialized to 0; t_L is initialized to 1.

The t values of the intersection points between the line and the clip edges are determined.

For each t value:

Classify it as "potentially entering" (PE) or "potentially leaving" (PL). It is potentially entering if P_0P_1 is (roughly) in the direction opposite to the normal; that is, if $(P_1 - P_0) \bullet N < 0$. (Note that this is the denominator of the expression used to compute t.) It is potentially leaving if $(P_1 - P_0) \bullet N > 0$, indicating that the line P_0P_1 is pointing (roughly) in the same direction as the normal.



fo

if the line is PE at that intersection point {

if $t > t_L$ then the line lies entirely outside the clip polygon, so it can be rejected;

else $t_E = \max(t,t_E)$;

}

else if the line is PL at that intersection point {

if $t < t_E$ then the line lies entirely outside the clip polygon, so it can be rejected;

else $t_L = \min(t,t_L)$;

}

}

if the line has not been rejected, then t_E and t_L define the endpoints of the clipped line.

The Liang-Barsky version of the algorithm recognizes athat if the clipping polygon is an upright polygon bounded by x_{xmin}, x_{max} , y_{min}, and y_{max}, the calculations can be simplified. The normal vectors are (1,0), (0,1),(-1,0), and (0,-1). The points P_e can be chosen as $(x_{max},0)$, $(0,y_{max})$,$(x_{min},0)$, and $(0,y_{min})$. The values of $(P_1 - P_0)\bullet N$ are (x_1-x_0), (y_1-y_0), (x_0-x_1), and y_0-y_1). The t values at the intersection points are $(x_{max}-x_0)/(x_1-x_0)$, $(y_{max}x-y_0)/(y_1-y_0$ $(x_0 - x_{min})/(x_0-x_1)$, and $(y_0-y_{min})/(y_0-y_1)$.

Question

1. Explain Cyrus beck algorithm?

Notes:

LESSON 11
2D GEOMETRICAL TRANSFORMATION, MATRIX REPRESENTATIONS

Today's Topics Covered in the Unit

- **Introduction to the Unit**
- **2d geometrical Transformations**
  1. Translation
  2. Rotation
  3. Scaling
- **Matrix Representations for2d**
- **3d geometrical Transformations**
- **Matrix Representations for 3d**
- **Viewing in 3d**
- **Projections**
- **Perspective projections**
- **Parallel projections**

Learning Objectives

Upon completion of this chapter, the student will be able to :

- Explain 2d geometrical Transformations
- Matrix Representations for 2d
- Composition of 2d Transformations

Today's Topics

- Introduction to the unit
- 2d geometrical Transformations
  - Translation
  - Scaling

Geometric Transformations

Introduction

Geometric transformations are used for several purposes in computer graphics:

- Window-to-viewport mapping (world coordinates to screen coordinates)
- Constructing object models
- Describing movement of objects (for animation)
- Mapping 3D to 2D for viewing (projection)

The basic transformations are all linear transformations. The same types of transformation are used in 2D and 3D. We'll start by looking at 2D transformations, then generalize to 3D.

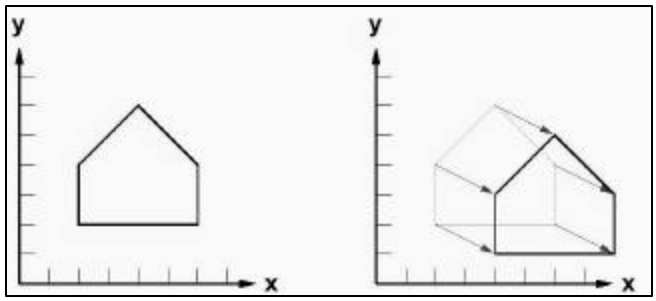
2D Transformations

Points in 2-dimensional space will be represented as column vectors:

We are interested in three types of transformation:

- Translation
- Scaling
- Rotation

Translation



Translation can be described algebraically by vector addition. Given a point P(x,y) and a vector T(dx,dy), a translation of P by T gives the point

P' = P+T = (x+dx,y+dy)

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} dx \\ dy \end{bmatrix}$$

Observe that by translating every point in a primitive by the same amount, we translate the entire primitive. In fact, we can translate an entire line by translating the endpoints, then connecting the translated endpoints.

Scaling

Scaling represents a stretching or shrinking with respect to the x-axis, the y-axis, or the origin. (The figure above shows a shrinking with respect to the origin.) Scaling with respect to the x- or y-axis distorts the shape of a primitive, in addition to altering its size.

Scaling can be represented algebraically by scalar multiplication. Scaling P(x,y) by a factor of s_x with respect to the x-axis gives P'(s_x x,y). Scaling P(x,y) by a factor of s_y with respect to the y-axis gives P'(x,s_y y). This operation can also be expressed by matrix multiplication:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} s_x & 0 \\ 0 & s_y \end{bmatrix} \bullet \begin{bmatrix} x \\ y \end{bmatrix}$$

As with translation, we can scale an entire figure by scaling each point in the figure. We can scale a line by scaling each endpoint, then forming the line between the scaled endpoints.

LESSON 12
2D GEOMETRICAL TRANSFORMATION, MATRIX REPRESENTATIONS (CONTD....)

Today's Topics

- **2d geometrical Transformations**
  - Rotation

Rotation

A point can be rotated through an angle about the origin using the equations:

$x' = x \cos\theta - y \sin\theta$

$y' = x \sin\theta + y \cos\theta$

In matrix form, we write

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$

That is,

$P' = R\,P$

where R is the rotation matrix. "

These equations can be derived as follows:

Let P(x,y) be a point. Express P in polar coordinates P(r, *f*). Then

$x = r \cos\phi$

$y = r \sin\phi$

Applying a rotation of θ about the origin simply adds θ to ϕ, giving

$x' = r \cos(\phi + \theta)$

$y' = r \sin(\phi + \theta)$

Applying the trig identities for addition of angles, we get

$x' = r(\cos f \cos q - \sin f \sin q) = r \cos f \cos q - r \sin f \sin q$

$= x \cos q - y \sin q$

$y' = r(\sin f \cos q + \sin q \cos f) = r \sin f \cos q + r \cos f \sin q$

$= y \cos q + x \sin q$

In graphics systems, points are usually represented using *homogeneous coordinates* . A point (x,y) in 2 dimensional space is represented by the triple (x,y,1); a point (x,y,z) in 3-dimensional space is represented by the quadruple (x,y,z,1).

Why?

Linear transformations of a vector space can be represented by matrices. A linear transformation can be applied to a point by multiplying the point (viewed as a column vector) by the matrix which represents the linear transformation. We would like to apply this to our three basic types of transformation (translation, scaling, and rotation).

But there's a problem.

Linear transformations of a vector space always map the origin to the origin. We can see this easily by seeing what happens when we multiply a 2 x 2 matrix by the (0,0) column vector. However, a translation by the vector (dx,dy) maps the origin to the point (dx,dy). Therefore, translation cannot be a linear transformation, and cannot be represented by matrix multiplication. (Scaling and rotation *are* linear transformations.)

So, what to do?

Consider the 2-D case. The set of points in the plane is a vector space. We can embed the plane in 3-D space, as follows:

Let $A = \{ (x,y,z) \text{ in } R^3 \mid z = 1 \}$. This set of points forms a plane in R^3 which is known as the *standard affine 2-dimensional space in R^3*. It satisfies the property that

{ u - v | u and v belong to A } is a vector subspace of R^3. This is the definition of an affine space.

Now, consider the matrix

$$\begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix}$$

This matrix actually represents a type of linear transformation called a shear transformation, when applied to R^3. But when restricted to the affine space A, it performs a translation by the vector (dx,dy).

As a result, if we adopt this view of the plane as being the affine space A, all of our basic geometric transformations are linear transformations which can be represented by matrices.

Why is this important?

Consider the notion of composition of functions. Suppose we want to apply two linear transformations f_1 and f_2 to a point p, one after the other. That is, we want to compute

$p' = f_2(f_1(p))$

We would do this by first multiplying p by the matrix representation of f_1, then multiplying the result by the matrix representation of f_2:

$P' = M_2 \times (M_1 \times P)$

Because the matrix multiplication is associative, this is equivalent to

$P' = (M_2 \times M_1) \times P$

This means that we can first form the product of M_2 and M_1, then multiply the resulting matrix by P to arrive at P'.

In other words, *the matrix representation of (f_2* composed with *f_1) is ($M_2 \times M_1$).*

This is extremely important for a typical graphics application, in which the same geometric transformation is often applied to a large number of vertices in a scene. To apply m transformations to a set of n points, we first perform (m-1) matrix

multiplications on the m transformation matrices, then multiply the result by each of the n points individually.

Note: An alternative way to interpret homogeneous coordinates is that they represent lines in R^3 which pass through the origin. With this interpretation, two triples $P_0(x_0,y_0,z_0)$ and $P_1(x_1,y_1,z_1)$ are equivalent if they are on the same line through the origin; that is, if the coordinates of P_1 are a scalar multiple of the coordinates of P_0. In this scheme, each point has infinitely many representations using homogeneous coordinates. By convention, we choose the representation with z = 1.

In summary, then, can represent translation, scaling, and rotation as matrices applied to homogeneous coordinates, as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & dx \\ 0 & 1 & dy \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \text{ that is, } P' = T \bullet P$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \text{ that is, } P' = S \bullet P$$

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \bullet \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \text{ that is, } P' = R \bullet P$$

Notes:

LESSON 13
COMPOSITION OF 2D TRANSFORMATION, WINDOW TO VIEW PORT TRANSFORMATION

Today's Topics

- **2d Geometrical Transformations**
  - Affine transformation
  - Shear transformation

Transformations

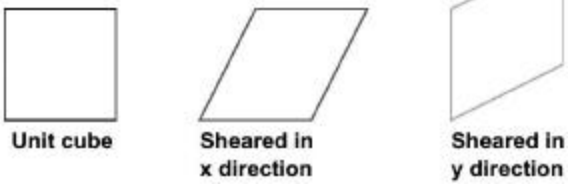
The product of an arbitrary sequence of translations, scalings, and rotations, is called an affine transformation. Affine transformations preserve the parallelism of lines. An affiine transformation can be expressed as a matrix in the form

$$\begin{bmatrix} a_{11} & a_{12} & t_x \\ a_{21} & a_{22} & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Translation and Rotation are examples of *rigid-body* transformations, which preserve both angles and lengths. They do not alter the size or shape of an object, only its position and orientation. The matrix for a rigid-body transformation has the properties:

1. $a_{11}^2 + a_{12}^2 = 1$
2. $a_{21}^2 + a_{22}^2 = 1$
3. $a_{11}a_{21} + a_{12}a_{22} = 0$
4. $a_{11}a_{22} - a_{12}a_{21} = 1$

Another type of transformation which is sometimes of interest is a *shear* transformation. A shear transformation has a simple geometric interpretation as a skewing of the coordinate system.



A shear in the x direction can be expressed using the following matrix:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & a & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

A shear in the y direction can be expressed using the following matrix:

Notes:

LESSON 14
COMPOSITION OF 2D TRANSFORMATION, WINDOW TO VIEW PORT TRANSFORMATION (CONTD....)

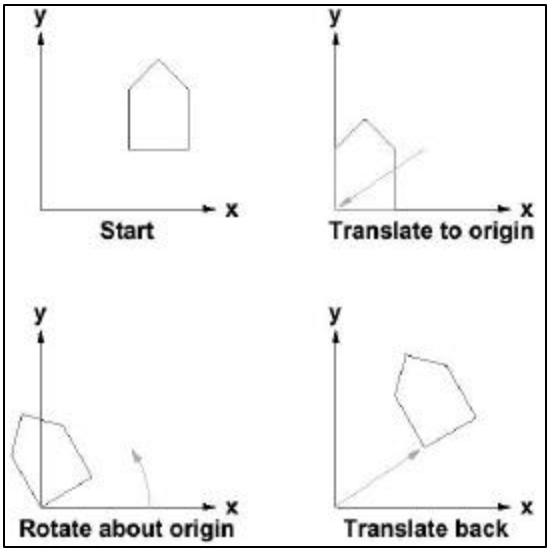
Today's Topics

- **2d geometrical Transformations**
  - Composition transformation
  - Window to viewport transformation

Composition of Transformations

Often, we need to create the desired transformations by composition of the basic ones. (In fact, shear transformations can be constructed by composing rotations with scaling.) One of the simplest examples is the rotation by angle q of an object about a point P_1 (x_1,y_1) other than the origin. This can be accomplished in three steps:

1. Translate so that P_1 goes to the origin
2. Rotate about the origin
3. Translate so that P_1 returns to its original position



The transformation matrix is computed by multiplying the matrices

$T(x_1,y_1)R(\theta)T(-x_1,-y_1)$

Question: Why are the matrices written in this order? Is the order important? "

Of the six combinations of primitive operations, the following pairs commute.

- Translation with translation.
- Scaling with scaling.
- Rotation with rotation.
- Rotation with uniform scaling ($s_x = s_y$).

The following do not

- Translation with scaling.
- Translation with rotation.
- Rotation with non-uniform scaling.

Window to Viewport Transformations

It is often convenient to specify primitives in a meaningful world-coordinate system (miles, microns, meters etc.) In general, this will not match up with the screen coordinate system, either in origin or units.

The mapping from world coordinates to screen coordinates is another example of a linear transformation. How can this linear transformation be specified? One way is for the programmer to provide a transformation matrix explicitly. A simpler way (from the point of view of the programmer, anyway) is for the programmer to specify matching rectangular windows - a world-coordinate window and a screen coordinate viewport. The graphics package can determine the transformation matrix from this information.

The final transformation matrix can be determined by composing the matrices for the following transformations:

1. Translate the world-coordinate window to the origin.
2. Rotate the window to match the orientation of the screen window (if necessary).
3. Scale the window to match the size of the viewport.
4. Translate the window to the screen

Notes:

LESSON 15
MATRIX REPRESENTATION OF 3D GEOMETRICAL TRANSFORMATION

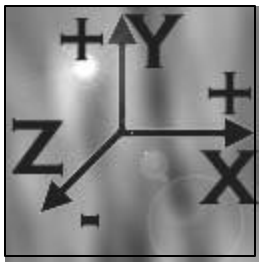
Today's Topics

- **3d Geometrical Transformations**
  - Matrix representation for 3d

An introduction to 3D

Ok so here it starts... with the coordinate system. You probably know that in 2-D, we usually use René Descartes's Cartesian System to identify a point on a flat surface. We use two coordinates, that we put in parentheses to refer to the point: (x, y) where *x* is the coordinate on the horizontal axe and *y* on the vertical one. In 3 dimensions, we add an axe called *z*, and usually we assume it represents the depth. So to represent a point in 3D, we use three numbers: *(x, y, z)*. Different cartesian 3D systems can be used. But they are all either *Left-Handed* or *Right-Handed*. A system is *Right-Handed* when pointing your index in the positive Y direction and your thumb in the positive X direction, your fingers are curled toward the positive Z direction. On the other hand, (hehe) a system is Left-Handed when your fingers are curled toward the negative Z direction. Actually, you can rotate these systems in any directions and they will keep these caracteristics. In computer graphics, the typical system is the Left-Handed so we'll use it too. So for us:

- X is positive to the right
- Y is positive going up
- Z is positive disappearing into the screen



Vectors

What is a vector exactly? In a few words, it's a set of coordinates... But if you get into more specific details, a vector can be a lot more. Let's start with a 2D vector, of the form (x, y): so let's talk about the vector P (4,5). (Usually, we put some weird arrow with only one side on top of the P, so it looks more like a hook). We can say that the vector P represent the point (4,5), or more likely that it is an arrow pointing from the origin to the point (4,5), having a specific direction and length. By the way, when we're talking about the length of a vector (also called the module), we talk about the distance from the origin to the point, and it's noted | P |. We compute the length of a 2D vector with the formula:

$$| P | = sqrt(x^2 + y^2)$$

Here's an interesting fact: In 1D (where a point is on a single axe), the square root of the square of a number corresponds to its absolute value, whence the | | symbol for the absolute value's notation.

Now let's jump to 3D vectors: our friend will be P(4, -5, 9). The length will be:

$$| P | = sqrt(x^2 + y^2 + z^2)$$

and it is represented by a point in Cartesian 3D space, or rather by an arrow pointing from the origin of the system to the point. We'll learn more about vectors when we'll talk about operations.

Matrices

I'll try to make this clear and simple at first: a matrix is a two-dimensional array of numbers Probably all matrices we'll use in this site we'll be 4 by 4. Why 4 by 4? Because we are in **3** dimension and because we need an additional column and an additional row to make the calculations work. In 2D we would need 3x3 matrices. This means that in 3D, you have 4 numbers horizontally, and 4 vertically, 16 in total. Look at a sample matrix:

A 4x4 identity matrix			
1	0	0	0
0	1	0	0
0	0	1	0
0	0	0	1

It's called the identity matrix, because when another matrix is multipled by this one, it isn't changed in any way. Now, just for fun, here's another example of what a matrix can look like:

A weird sample matrix			
10	-7	22	45
sin(a)	cos(a)	34	32
-35	28	17	6
45	-99	32	16

Operations on Vectors and Matrices

So you've found all you've read here pretty easy and are wondering when you will learn something? Or you're even asking yourself what is the link between all this information and 3D graphics? Here everything changes, you will now learn facts that are the foundation of 3D transformations and of a lot of other concepts. It's still mathematical stuff though... We'll talk about operations on Vectors and Matrices: the sum and different type of products. Let's start with the addition of two vectors:

$$(x_1, y_1, z_1) + (x_2, y_2, z_2) = (x_1 + x_2, y_1 + y_2, z_1 + z_2)$$

Quite simple heh? Now the product of a scalar by a vector:

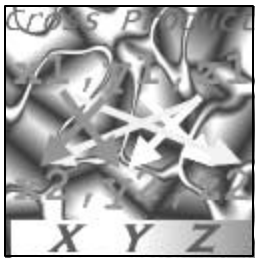
$$k \cdot (x, y, z) = (kx, ky, kz)$$

Now a trickier one, called the dot product, doesn't get a vector as a result: $(x_1, y_1, z_1) \cdot (x_2, y_2, z_2) = x_1x_2 + y_1y_2 + z_1z_2$ Actually, the dot product of two vectors divided by the product of their modules, corresponds to the cosine of the angle between the vectors. So:

$$\cos(V \wedge W) = V \cdot W / (|V| * |W|)$$

Note that the "^" doesn't mean exponent this time, but the angle between the vectors! This application of the dot product can be used to compute the angle of a light with a plane so it will be discussed in greater details in the section about Shading.

Now a very weird one, the cross product.



$$(x_1, y_1, z_1) X (x_2, y_2, z_2) = (y_1z_2 - z_1y_2, z_1x_2 - x_1z_2, x_1y_2 - y_1x_2)$$

The cross product is very useful to compute the normal of a plane.

Ok, we've finished with the vectors. I'll begin with the sum of two matrices. It's pretty straightforward and similar to the sum of two vectors, so I won't write a big formula here. For every i which is a row in the matrices, and for every j which is a column in the matrices, you simply add the term (i, j) of the second matrix to the term (i, j) of the first one. I could write some big formula with weird looking big sigma symbols but I don't want to... We'll rather move to the most important principle in matrices, concerning 3D transformations: the product of two matrix. I will point right now the fact that M x N **** DOESN'T **** equal N x M. So here is the equation for multiplying two matrices, this time with the sigmas. You probably won't understand anything if you don't already know the principle, but it will get clear when you'll see the code in the tutorial about 3D transformations. Here it is:

A 4x4 matrix multiplication formula
If $\mathbf{A}=(a_{ij})_{4x4}$ and $\mathbf{B}=(b_{ij})_{4x4}$, then
A x B=

$\sum_{j=1}^{4} a_{1j}b_{j1}$	$\sum_{j=1}^{4} a_{1j}b_{j2}$	$\sum_{j=1}^{4} a_{1j}b_{j3}$	$\sum_{j=1}^{4} a_{1j}b_{j4}$
$\sum_{j=1}^{4} a_{2j}b_{j1}$	$\sum_{j=1}^{4} a_{2j}b_{j2}$	$\sum_{j=1}^{4} a_{2j}b_{j3}$	$\sum_{j=1}^{4} a_{2j}b_{j4}$
$\sum_{j=1}^{4} a_{3j}b_{j1}$	$\sum_{j=1}^{4} a_{3j}b_{j2}$	$\sum_{j=1}^{4} a_{3j}b_{j3}$	$\sum_{j=1}^{4} a_{3j}b_{j4}$
$\sum_{j=1}^{4} a_{4j}b_{j1}$	$\sum_{j=1}^{4} a_{4j}b_{j2}$	$\sum_{j=1}^{4} a_{4j}b_{j3}$	$\sum_{j=1}^{4} a_{4j}b_{j4}$

And if $\mathbf{AxB}=(c_{1k})_{4x4}$ then we can write this on one line:

$$c_{ik} = S_{4,\, j=1}\, a_{ij}b_{jk}$$

Now you should be able to try multiplying some matrix by an identity matrix to understand how it works. Then after all these separated discussions about vectors and matrices, we's multiply them together! So here's the formula to multiply a 3D vector by a 4x4 matrix (you should already have guessed that the result will be another 3D vector), if $\mathbf{B}=(b_{ij})_{4x4}$:

$$(a_1, a_2, a_3) \; x \; B = (S\, a_i b_{i1} + b_{4,1},\; S\, a_i b_{i2} + b_{4,2},\; S\, a_i b_{i3} + b_{4,3})$$

with ***3, i=1*** as parameters for the sums.

That's it for the operations on vectors and matrices! It's getting harder, heh? From now on, the link between the code and the maths will be more visible, with transformations...

Transformations

You've surely already seen formulas like:

$$t_{(tx,\, ty)}: (x, y) \Longrightarrow (x + tx, y + ty)$$

This was the equation of a translation in a 2D Cartesian system. Now let's check the scaling equation:

$$s_{(k)}: (x, y) \Longrightarrow (kx, ky)$$

Makes sense heh? A much harder one, the rotation, where trigonometry makes its entry in 3D graphics:

$$r(q): (x, y) \Longrightarrow (x \cos(q) - y \sin(q),\; x \sin(q) + y \cos(q))$$

These were for 2D, but in 3D they stay pretty much the same. You simply add the coordinate ***z*** and the parameter tz for the translation. For the scaling, you simply multiply ***z*** by ***k*** (or you can use three diffrent scalings for every coordinates, like in the scaling matrix below). For the rotation, you keep the same formula, let ***z*** stays the same, and it gives you the rotation around the ***z axis***. Because two other rotations are added in 3D (around the ***x*** and ***y axis***). I could write all this 3D transformation the same way I did in 2D, but instead we'll use a much cleaner way, (that will show you the point of all this chapter) vectors and matrices! So you have your vector (x, y, z) as above in 2D, and several matrices of trasformation, one for each type. Then we will multiply the matrices by the vector and the resulting vector will be pointing to the transformed point. (In the next chapter, we will multiply every matrices together, to get what we will called the global transformation matrices, then multiply it by the source vector to get the destination in only one operation!). So let's show you all these 3D transformation matrices:

Matrix for a 3D translation of (tx, ty, tz)			
1	0	0	0
0	1	0	0
0	0	1	0
tx	ty	tz	1

Matrix for a 3D scaling of (sx, sy, sz)			
sz	0	0	0
0	sy	0	0
0	0	sx	0
0	0	0	1

Matrix for a 3D rotation around the x axis of θ			
1	0	0	0
0	cos(θ)	sin(θ)	0
0	-sin(θ)	cos(θ)	0
0	0	0	1

Matrix for a 3D rotation around the y axis of θ			
cos(θ)	0	-sin(θ)	0
0	1	0	0
sin(θ)	0	cos(θ)	0
0	0	0	1

Matrix for a 3D rotation around the z axis of θ			
cos(θ)	sin(θ)	0	0
-sin(θ)	cos(θ)	0	0
0	0	1	0
0	0	0	1

So this concludes the part about transformations. You can apply any transformation to a 3D point with these matrices. In the next chapter, we will implement the code for matrices, vectors and for transforming 3D points. But before moving to the coding part, I want to discuss a bit planes and normals...

Planes and Normals

A plane is a flat, infinite surface, oriented in a specific direction. You can define a plane with the famous equation:

$$Ax + By + Cz + D = 0$$

where A, B, C are what we called the ***normals*** of the plane, and D is the distance from the plane to the origin. So what is a normal? It's a vector perpendicular to a plane. We compute the normal of a plane by doing the cross products of the two edges of the plane. To define these edges, we need three points. If $\mathbf{P}_1$ is our fisrt point, $\mathbf{P}_2$ our second, and $\mathbf{P}_3$ our third, and if they are counter-clockwise, treating them as vectors we can write:

$$Edge_1 = P_1 - P_2$$

and

$$Edge_2 = P_3 - P_2$$

and then compute the normal:

$$Normal = Edge_1 \; X \; Edge_2$$

What about the D component of the equation? We simply isolate D, plot the values of any of the three point in the equation, and we get it:

$D = -(Ax + By + Cz)$

or

$D = -(A \cdot P_1.x + B \cdot P_1.y + C \cdot P_1.z)$

or even trickier:

$D = -Normal \cdot P_1$

But to compute the A, B, C components (because sometimes, we need themselves, not the normals), you can simplify all these operations with these equations:

$$A = y_1(z_2 - z_3) + y_2(z_3 - z_1) + y_3(z_1 - z_2)$$
$$B = z_1(x_2 - x_3) + z_2(x_3 - x_1) + z_3(x_1 - x_2)$$
$$C = x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)$$
$$D = -x_1(y_2z_3 - y_3z_2) - x_2(y_3z_1 - y_1z_3) - x_3(y_1z_2 - y_2z_1)$$

Notes:

LESSON 16
MATRIX REPRESENTATION OF 3D GEOMETRICAL TRANSFORMATION (CONTD…)

Today's Topics

· **3d geometrical Transformations**

- Composition of 3d transformations

More detail on 3D by prof . J. Donaldson (University of Berlin)

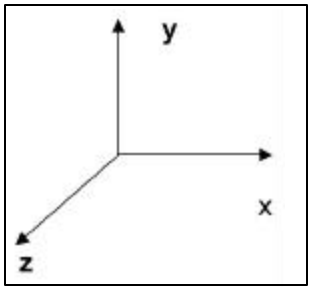
Three-dimensional Transformations

Just as 2D transformations can be represented using 3 x 3 matrices and homogeneous coordinates, 3D transformations can be represented using 4 x 4 matrices and a homogeneous representation for a 3-vector.

We use the same basic transformations, but in 3D:

- 3D translation (by a 3-component displacement vector).
- 3D scaling (3 values, along the three axes).
- 3D rotation (by an angle about any of the three axes).

For rotation, the "handedness" of the coordinate system matters. In a right-handed system, with the usual layout of the x- and y-axes on the surface of the viewing screen, the positive z-axis points from the screen to the viewer. The negative z-axis points into the screen. For a right-handed system, a 90 degree positive (counterclockwise) rotation about the z axis looking from +z towards the origin, takes +x into +y. (In general, positive values of q cause a counterclockwise rotation about an axis as one looks inward from a point on the positive axis toward the origin.)



Matrix Representations for 3D Transformations

The homogeneous representation for the point (x, y, z) is (x, y, z, 1).

Translation is represented by the matrix:

$$T(dx, dy, dz) = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling has the form:

$$S(s_x, s_y, s_z) = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

There are three rotation matrices, one for rotation about each axis.

For rotation about the z-axis, the matrix is just an extended version of the 2D rotation matrix:

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

The x-axis rotation matrix is:

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

The y-axis rotation matrix is:

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

The transformation matrix for a rotation about an arbitrary axis can be decomposed into a sequence of rotations about the three coordinate axes. (This is a theorem of Euler.) The strategy to do this is similar to what was used to rotate about an arbitrary point in 2D space. To rotate about an arbitrary direction vector u (which can also be interpreted as a line through the origin):

Perform a y-rotation that puts u in the xy-plane
Perform a z-rotation that puts u in the x-axis
Perform the desired rotation about the x-axis
Apply the inverse of the z-rotation
Apply the inverse of the y-rotation

OpenGL performs this calculation for us. The function glRotate*(θ,x,y,z) generates the transformation matrix to rotate by q about the direction vector with components x, y, and z. For example, glRotatef(30.0,0.0,1.0,0.0) would generate the transformation matrix for a rotation by 30 degrees around the y-axis.

The general form of an affine transformation obtained by multiplying a sequence of translation, scaling, and rotation matrices together is:

$$\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_x \\ r_{21} & r_{22} & r_{23} & t_y \\ r_{31} & r_{32} & r_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The r_{ij} entries represent a combination of rotation and scaling, and the t_i entries are a translation. Such matrices are generally invertible (unless a scale to 0 on some axis or projection is performed). As before, a rigid transformation is composed from translation and rotation, but no scaling operations, and preserves angles and lengths. The upper left 3 x 3 submatrix of such a matrix is orthogonal. (That is, the rows are orthogonal unit vectors and the determinant is 1.)

Composition of 3D Transforms

3D transformations can be composed in the same way as the 2D transformations. Translation, scaling, and rotation about a single axis commute with themselves, as do uniform scaling and any complex rotation. Other combinations, including rotations about different axes do not commute. 3D transformations can be determined in various ways.

Transformations as Changes of Coordinate System

So far, we have thought of transformations as moving objects within a coordinate system. An equivalent point of view is to consider transformations as changing the coordinate system that an object is described in. Transforming a coordinate system means that all the points in the space get relabeled (as opposed to relabeling the object points).

For example, the translation x' = x + dx can either be viewed as moving an object along the x-axis by dx, or as placing the object in a new coordinate system, whose origin is displaced by an amount -dx along the x axis. A general rule is that transforming an object by Q is equivalent to transforming the coordinate system by Q^{-1}. (The OpenGL view is that the coordinate system is transformed by Q *before* the object is drawn.)

This is useful because it is often convenient to describe objects within their own "personal" (local) coordinate system. These objects may then be placed, at various locations, scales, and orientations, in a global (world) coordinate system.

How does this work in OpenGL?

What happens when a program draws a geometric primitive using a glBegin()...glEnd() block? Each vertex in the block is subject to a sequence of transformations represented by matrices, beginning with the modelview matrix, followed by the projection matrix. We'll talk more about the projection matrix next week. Today we will just consider the modelview matrix.

The modelview matrix is intended to perform two functions. The first is modeling. It is assumed that each object is defined in its own object coordinate system. Often the object is placed so that its center of mass or some other prominent point is placed at the origin. The modeling part of the transformation allows the object to be placed, with a certain position, scale, and orientation, in a scene which may contain other objects. The coordinate system in which the scene is described is called the world coordinate system. So the modeling transformation can be interpreted as a change of coordinates from object coordinates to world coordinates. (Of course, this means that each object may need to have its own modeling transformation.) In OpenGL, modeling is usually a sequence of glRotate, glScale, and glTranslate commands, although the program may create a transformation matrix explicitly.

Once the world has been created, the next step is to position the viewer. This is done through the view transformation. The view transformation translates world coordinates to what are called eye or camera coordinates. The OpenGL Utility library (GLU) contains a function which establishes a viewing matrix: gluLookAt. This function has 9 parameters, representing a point at which the viewer is located, a point in the direction that the viewer is looking, and a vector indicating which direction is "up" from the point of view of the viewer.

The modelview matrix is really the product V x M of a viewing matrix and a modeling matrix. Note that because OpenGL applies the matrix using postmultiplication by a column vector, the first operation performed is the one on the right; hence we write V x M to indicate that the modeling component of the matrix is applied first.

OpenGL keeps the modelview matrix as part of its state. Whenever a vertex is drawn, it is multiplied by the current modelview matrix. It is the programmer's responsibility to put the proper values into the modelview matrix.

When one of the calls glRotate, glScale, or glTranslate is made, the modelview matrix is postmultiplied by the transformation matrix indicated by the call. As a result, the last transformation multiplied is the first one applied to each geometric primitive which follows. This is somewhat counterintuitive, so the programmer must be careful to specify transformations in the proper order.

A typical sequence of calls would be:

```
glMatrixMode(GL_MODELVIEW);
                    // Select Modelview as the matrix currently being operated on
glLoadIdentity();   // Initialize the modelview matrix as the identity matrix
gluLookAt(0.0,0.0,0.0,0.0,0.0,-1.0,0.0,1.0,0.0);
                    // Multiply by a viewing matrix to establish the position of the camera
glTranslatef(0.5,0.3,0.4);      // Multiply by a translation matrix
glRotatef(30.0,1.0,0.0,0.0);    // Multiply by a rotation matrix
glScalef(1.2,1.2,1.2);          // Multiply by a scaling matrix
glBegin();                      // Draw the object
.
. glEnd();
```

The effect of this is to take an object, scale it, rotate it, and translate it to a position in the world coordinate system, then define a position from which a camera can view it. However, the sequence of operations makes more sense if viewed as a series of transformations of the coordinate system prior to drawing the object.

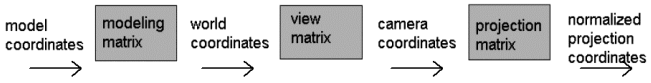
LESSON 17
VIEWING IN 3D

Today's Topics

- **3D Viewing**

3D Viewing

Problem: Given a 3D world, described in a 3D world coordinate system, how do we specify a view of this world? And, given such a specification, how do we construct the transformation matrices which will be used to transform points in the world to points on the viewing screen? Recall our conceptual view of the rendering pipeline:



We have seen how the modeling matrix can be constructed using translations, rotations, and scalings. Now we will consider the view and projection matrices.

The specification of a view is based on the following terminology:

1. view plane - the plane onto which an image is projected (same as projection plane or image plane)
2. VRP (view reference point) - a point chosen on the view plane
3. VPN (view plane normal) - a vector normal to the VRP
4. VUP (view up vector) - a vector indicating the orientation of the viewer (to the viewer, which direction is up?)
5. VRC (view reference coordinates) - a 3D coordinate system from the point of view of the viewer. Also called camera coordinates or eye coordinates.
6. view volume - a three-dimensional solid which determines what part of a scene will be visible to the user. Analogous to the notion of a clipping region in two dimensions

The VRC is constructed as follows:

1. VRP is the origin.
2. VPN is one of the coordinate axes of the system, called the n-axis.
3. VUP forms a second axis, called the v-axis.
4. The third axis, the u-axis, is obtained from the cross product of n and v, forming a right-handed coordinate system.

The view matrix maps world coordinates to the view reference coordinates, so that the z-axis is mapped to the n-axis, the y-axis to the v-axis, and the x-axis to the u-axis. In OpenGL, the default view matrix is the identity. In this default case the viewer is positioned at the origin, looking in the negative z direction, oriented so that the up vector is in the direction of the y-axis.

The viewing matrix which performs this mapping can be constructed by a combination of translation and rotation, as follows:

Perform a translation T(-VRP) to translate the VRP to the origin.
Perform an x-rotation so that VPN is rotated into the x-z plane.
Perform a y-rotation so that VPN is rotated into the negative z-axis.
Perform a z-rotation so that the projection of VUP onto the x-y plane is rotated into the positive y-axis.

The result is a matrix V = R x T. (As usual, the first operation to be applied is on the right.)

The upper left 3 x 3 portion of the R matrix can also be viewed as a list of three row vectors R_x, R_y, R_z, where

$R_z = -VPN / || VPN ||$

$R_x = (VUP \times R_z) / || VUP \times R_z ||$

$R_y = R_z \times R_x$

In OpenGL, the view transformation can be constructed by the user in one of two ways:

- by using the same translate, scale, and rotate functions that were used in constructing the modeling matrix, or
- by using the function gluLookAt.

gluLookAt takes 9 parameters, which define two points and a vector:

- the first point identifies the position of the camera or eye (effectively the VRP)
- the second point identifies a point in the direction that the camera is aimed (VPN is the vector from the VRP to this point)
- the vector is the up vector (VUP)

The View Volume

The view volume is a window on the world, which sets bounds on what part of the world will be projected onto the view plane. The idea is that some objects in the scene are too far to the left or right, or too far up or down, to be within the field of view of the viewer. The shape of the view volume depends on whether the projection to be used is a parallel or perspective projection.

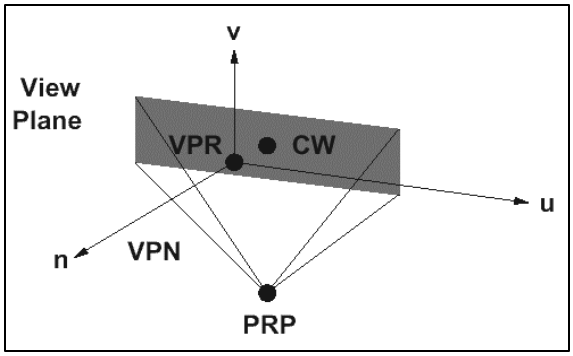
For a parallel projection, the view volume is a right parallelepiped bounded by the planes $x = x_{max}$, $x=x_{min}$, $y=y_{max}$, and $y=y_{min}$. For a perspective projection, the view volume is a semi-infinite pyramid whose apex is the center of projection.

In addition, it is customary to set limits on the z-coordinates in the view volume by defining front and back clipping planes. The front clipping plane is $z=z_{max}$ and the back clipping plane is $z=z_{min}$. This truncates the parallel view volume to a right parallelepiped of finite dimension and the perspective view volume to a frustum (that is, a pyramid with the top sliced off).

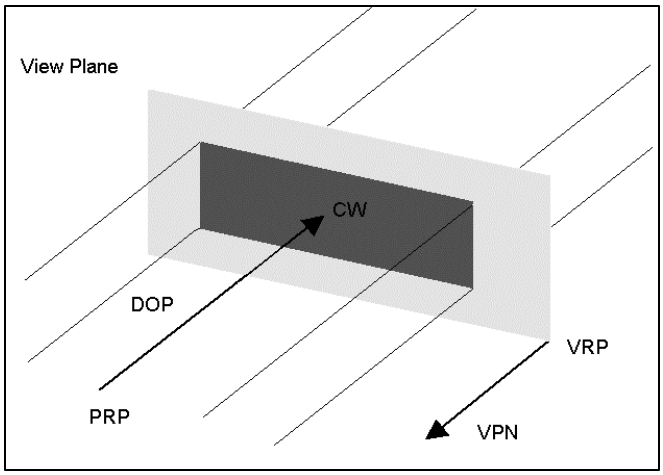
We will see that defining the view volume is an important step in constructing the projection transformation.

The view volume is defined formally within the view reference coordinate system as follows:

Choose a rectangular viewing window in the uv-plane. (Note: this rectangle need not be centered at the origin.) Then define a Projection Reference Point (PRP). In the case of a perspective projection, this point is the center of projection. The following diagram illustrates the pyramid-shaped view volume for a perspective projection:



In the case of a parallel projection, a vector from the PRP to the center of the viewing window (CW) is the direction of projection (DOP). The view volume is then an infinite parallelepiped as shown in the following diagram (which illustrates an orthogonal parallel projection):



Notes:

LESSON 18
PROJECTION

Topics in the Section

- **What is Projection**
- **What is Perspective Projection**
- **What is Parallel Projection and its different types**

Learning Objectives

Upon completion of this chapter, the student will be able to :

- Explain what is Projection
- Explain what is Perspective Projection
- Explain what is Parallel Projection

Today's Topics

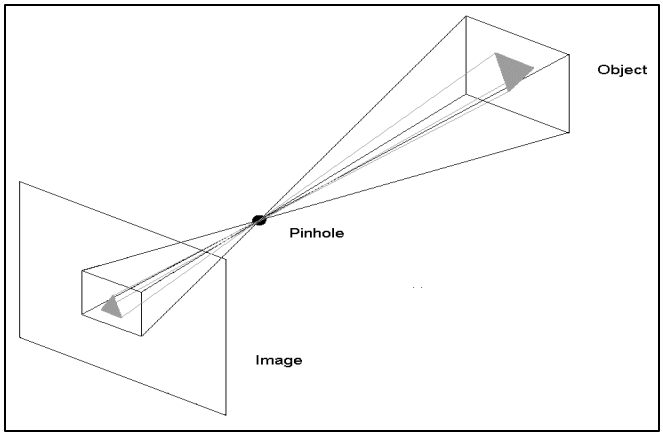
- **Projection an Introduction**

Projections Introduction

We are interested in how projections work in the n computer graphics. The basic problem, of course, is that the viewing screen is a two-dimensional screen. This requires the use of a transformation called a *projection*, which maps a space to a space of lower dimension. Many types of projection can be defined; some of these are of interest in computer graphics.

We begin by describing the model of a synthetic camera, which is useful in describing how a projection works.

Consider a pinhole camera:



- Rays of light from an object in front of the pinhole pass through the pinhole and form an image on a film behind the pinhole.
- All the rays, called *projectors*, intersect at the pinhole point, which is called the *center of projection*.
- A two-dimensional image is formed on the film behind the pinhole. This is called the *image plane* or *projection plane*.
- The image is upside-down.

We make some observations:

- The image that is formed on the projection plane is the same image we would get if we placed the projection plane in front of the pinhole, and used all the same projectors, except it is upside-down. It is customary to consider the projection plane as being on the same side of the center of projection as the model.
- Consider what would happen if we left the model and image plane stationary, but moved the center of projection away to infinity. In that case, the projectors all become parallel to each other.

The projections of interest in computer graphics are called geometric planar projections. They have the following properties:

- The projectors are straight lines that are parallel or intersect in a single point.
- The projection is formed as the intersection of the projectors with a plane (the projection plane).

The projection maps all the points on each ray to a single point, the point where that ray intersects the projection plane.

We are not interested in projections onto non-planar surfaces (for example, a sphere) or using curved projectors (like some map projections).

There are two basic classes of planar projections:

- Perspective: The projectors all pass through a point.
- Parallel: The projectors are parallel lines.

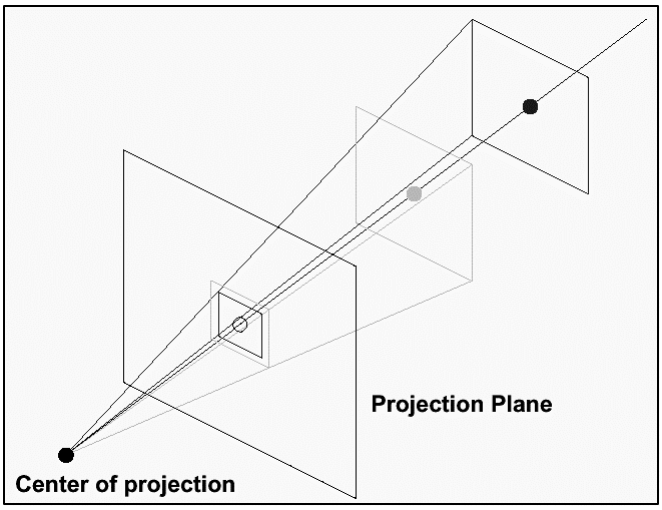
Notes:

LESSON 19
PROJECTION (CONTD…)

Today's Topics

- **Perspective Projection**
  - **Implementation of perspective projection**

Perspective Projection



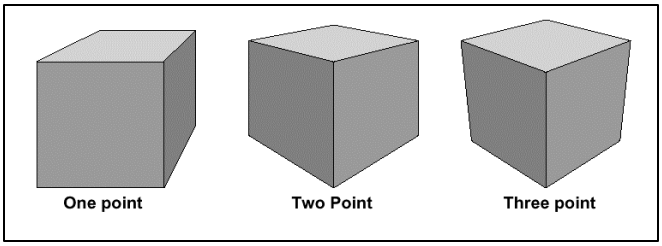
Determined by the placement of the center of projection and the projection plane. Or, in terms of our synthetic camera,

- The position of the camera lens
- The direction in which the camera is aimed (normal to the projection plane)

Basic characteristics of perspective projections:

- Similar to cameras and human vision.
- Look realistic.
- Does not preserve lengths or angles.
- Objects look smaller when farther away.
- Projected parallel lines intersect at a vanishing point (unless they are parallel to the projection plane).

The axis vanishing points are where lines parallel to x, y, and z axes appear to converge. A drawing is said to utilize 1, 2, or 3 point perspective, depending on how many of the principal axes intersect the projection plane.

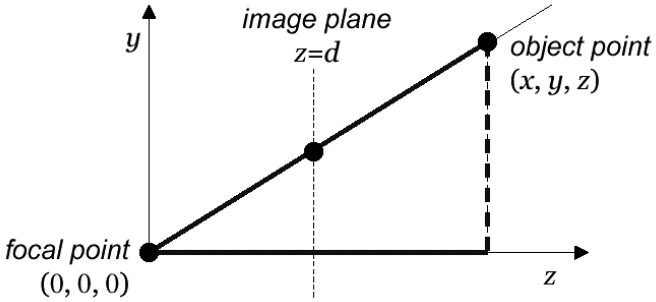


Implementation of Perspective Projection

Basic problem: How to formulate geometric transformations that perform projection.

Basic strategy: Start with simple case, and generalize by applying already known transformations.

Here, we assume that the projection plane is normal to z axis, at distance d, and the center of projection is at the origin.



The question then is: To what point (x_p, y_p, z_p) in the projection plane is the point (x,y,z) transformed by this perspective projection?

The answer can be determined easily using similar triangles.

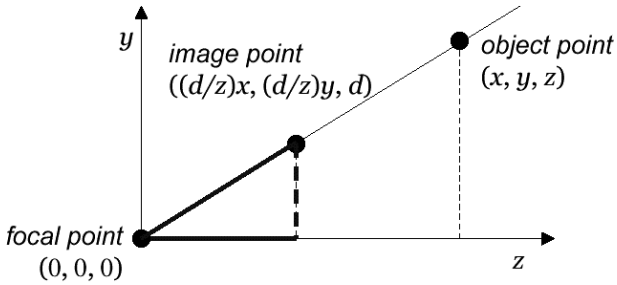
$y_p / d = y / z$

So

$y_p = dy / z = y / (z/d)$

Similarly,

$x_p = x / (z/d)$



Note how increasing z causes the projected values to decrease.

The transformation can be represented as a 4 x 4 matrix in homogeneous coordinates as follows. Consider the matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix}$$

When applied to the point (x,y,z,1), we obtain

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix}$$

Now, we homogenize the result. (Recall the interpretation of homogeneous coordinates as representing lines through the origin.) We obtain the projected point $(x_p,y_p,z_p,1)$.

So, projection can be expressed as a 4 x 4 matrix in homogeneous coordinates.

An alternative way to derive this transformation is to place the projection plane at z=0 and the center of projection at z=-d. (This will allow us to let d tend to infinity.)

In this case the similar triangles give us

$x_p / d = x / (d + z)$ ==> $x_p = x / (z/d + 1)$

$y_p / d = y / (d + z)$ ==> $y_p = y / (z/d + 1)$

$z_p = 0$

The transformation from (x,y,z) to (x_p,y_p,z_p) can be performed by the matrix

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1/d & 1 \end{bmatrix}$$

Letting d tend to infinity, we get the transformation matrix for orthographic parallel projection

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

which has the expected effect of setting the z-coordinate to 0.

Notes:

LESSON 20
PARALLEL PROJECTION

Today's Topics

- **Parallel Projection**
  - Orthographic projection
  - Multiview Projections

Parallel Projection

Parallel projections catagorize one of two major subclasses of planar geometric projections. Projections within this subclass have two characteristics in common. The first characteristic concerns the placement of the center of projection (PRP) which represents the camera or viewing position. In a parallel projection, the camera is located at an infinite distance from the viewplane (see Figure 3). By placing the camera at an infinite distance from the viewplane, projectors to the viewplane become parallel (the second characteristic of a parallel projection) in result forming a parallelepiped view volume. Only objects within the view volume are projected to the viewplane. Figure 3 shows the projection of line AB to the viewplane. In this case, the measurement of line AB is maintained in the projected line A'B'. While the measurements of an object are not preserved in all parallel projections, the parallel nature of projectors maintains the proportion of an object along a major axis. Therefore, parallel projections are useful in applications requiring the relative proportions of an object to be maintained.

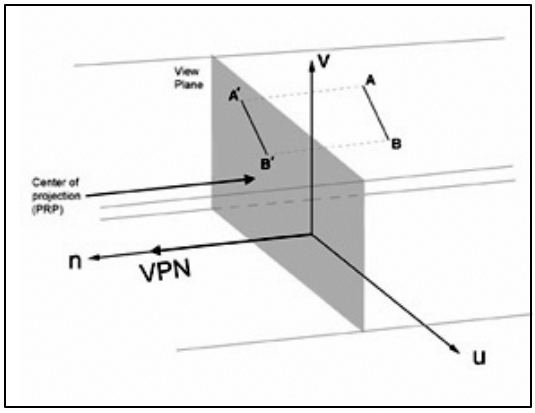
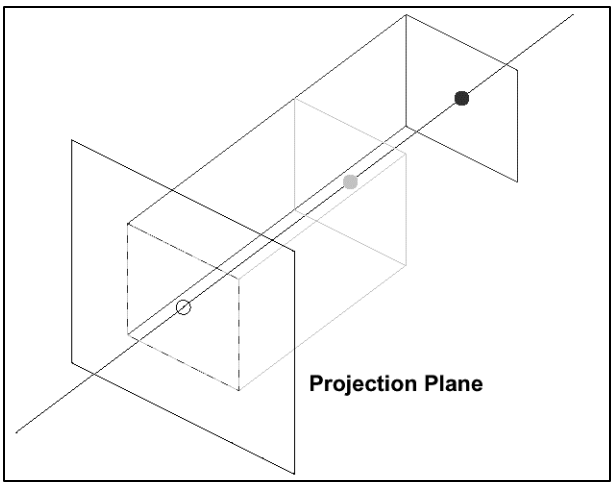


Figure 3 Parallel projection defined by the Center of Projection (PRP) placed at an infinite distance from the viewplane.

- Determined by a direction of projection. All projectors are parallel to this direction.
- Not as realistic as perspective, but certain measurements of distances and angles are preserved (so this type of projection is of interest to architects and designers).
- Two main classes: orthographic and oblique (each of which has further subclasses).

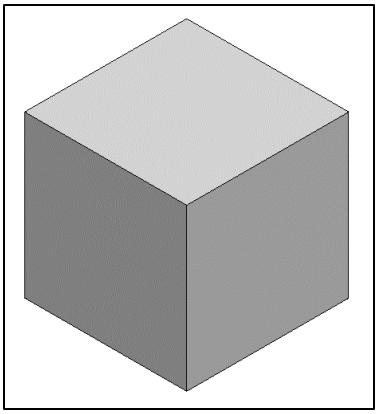
Orthographic Projection

- Projectors are perpendicular to projection plane.
- Front, top, side views: Named faces are parallel to projection plane.
- Can measure distances, angles on named faces.



Two common varieties are axonometric and isometric:

- Axonometric: All axes intersect plane at some angle, relative distances on axes can be measured with appropriate scale factor.
- Isometric: All three axes intersect projection plane at equal angles. Scale factors for axes are equal

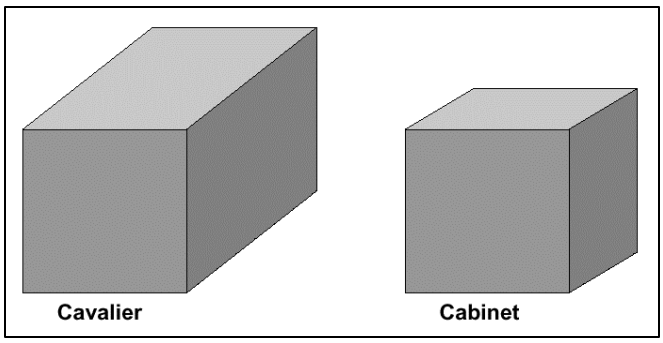


Oblique Projection

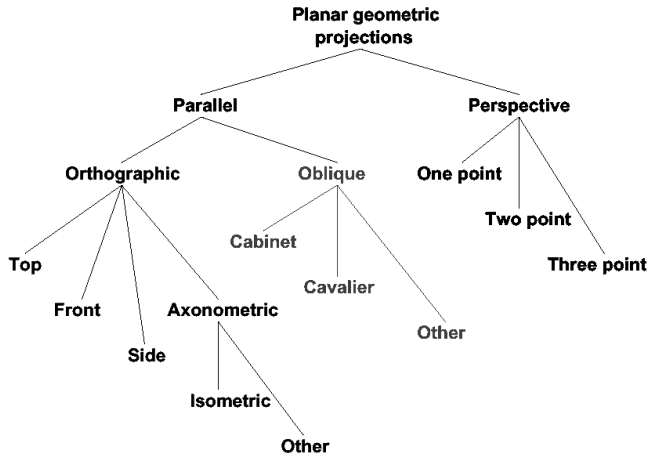
- Projectors form some angle other than 90 degrees with projection plane.
- Permits both an accurate face, and 3D structure to be displayed.

Two common varieties of oblique projection are:

- Cavalier: 45 degree projection, equal scales for z, x, and y. Tends to look a bit stretched.
- Cabinet: 63.4 (arctan2) projection, factor of 2 foreshortening of z relative to x and y. Looks more realistic.



The following tree shows the hierarchy of planar geometric projections:



Orthographic Projections

Orthographic projections are one of two projection types derived by subdivision of the parallel projection subclass. In addition to being parallel, projectors in an orthographic projection (shown in Figure 4) are also perpendicular to the viewplane (Hearn & Baker, 1996). Orthographic projections are further catorgorized as either multiview or axonometric projections, which are described below.

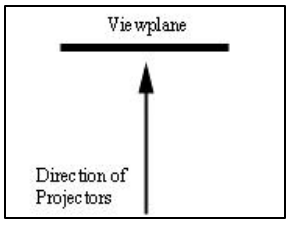


Figure 4 Direction of projectors for an orthographic projection.

Multiview Projections

A multiview projection displays a single face of a three-dimensional object. Common choices for viewing a object in two dimensions include the **front**, **side**, and **top** or planar view. The viewplane normal differs in the world coordinate system axis it is placed along for each of the multiview projections. In the top view, the viewplane normal is parallel with the positive y-axis in a right-handed system. Figure 5a illustrates a top or planar view of the three-dimensional building shown in Figure 5b. To project the top view of the 3-D object, the y-coordinates are discarded and the x- and z-coordinates for each point are mapped to the viewplane. By repositioning the viewplane normal to the positive z-axis and selecting the x-, and y-coordinates for each point, a front view is projected to the viewplane (Figure 5c). Likewise, a side view (Figure 5d) results when the viewplane normal is directed along the positive x-axis and the y- and z-coordinates of a three-dimensional object are projected to the viewplane. These projections are often used in engineering and architectural drawings (Hearn & Baker, 1996). While they do not show the three-dimensional aspects of an object, multiview projections are useful because the angles and dimensions of the object are maintained.

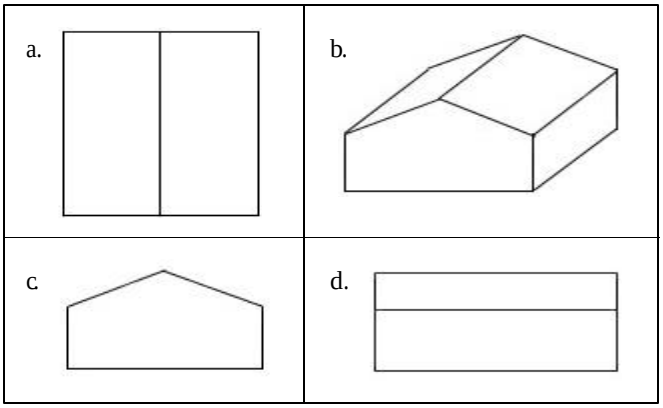


Figure 5 Two and three dimensional projections; a) top or planar view; b) three-dimensional view for reference; c) front view; and d) side view.

Notes:

LESSON 21
PARALLEL PROJECTION (CONTD…)

Today's Topics

- **Parallel Projection**
  - Axonometric Projections
  - Dimetric projections
  - Trimetric projections
  - Oblique Projections

A Little More in Detail

Axonometric Projections

Unlike multiview projections, axonometric projections allow the user to place the viewplane normal in any direction such that three adjacent faces of a "cubelike" object are visible. To avoid duplication of views displayed by multiview projections, the viewplane normal for an axonometric view is usually not placed parallel with a major axis (Hill, 1990). The increased versatility in the direction of the viewplane normal positions the viewplane such that it intersects at least two of the major axes. Lines of a three-dimensional object that are parallel in the world coordinate system are likewise projected to the viewplane as parallel lines. In addition, the length of a line, or line preservation, is maintained for lines parallel to the viewplane. Other receding lines maintain only their proportion and are foreshortened equally with lines along the same axes.

Axonometric projections are further divided into three classes that depend upon the number of major axes which are foreshortened equally (Hill, 1990). These axonometric views are defined as isometric, dimetric, or trimetric projections.

An **isometric projection** is a commonly used axonometric projection (Foley et al., 1996; Hearn & Baker, 1996). In this view, all three of the major axes are foreshortened equally since the viewplane normal makes equal angles with the principal axes. To satisfy this condition, the viewplane normal **n** = (nx, ny, nz) has the requirement that $|nx| = |ny| = |nz|$. This limitation restricts **n** to only eight directions (Foley et al., 1996). Figure 6 shows an isometric projection of a cube. Isometric projections scale lines equally along each axis, which is often useful since lines along the principal axes can be measured and converted using the same scale.

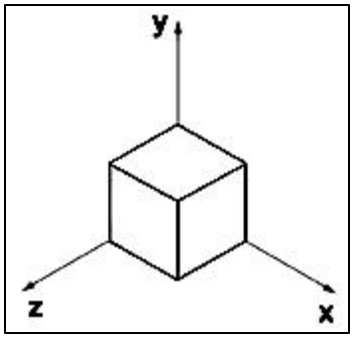


Figure 6 Isometric Projection

Dimetric projections differ from isometric projections in the direction of the viewplane normal. In this class of projections, **n** = (nx, ny, nz) is set so that it makes equal angles with two of the axes. Valid settings for a dimetric projection allow $nx = |ny|$, $nx = |nz|$, or $ny = |nz|$ (Hill, 1990). In this class, only lines drawn along the two equally foreshortened axes are scaled by the same factor. Figure 7 shows a dimetric projection of a cube. When the viewplane normal is set so that the viewplane is parallel to a major axis, line measurements are maintained in the projection for lines which are parallel to this axis.

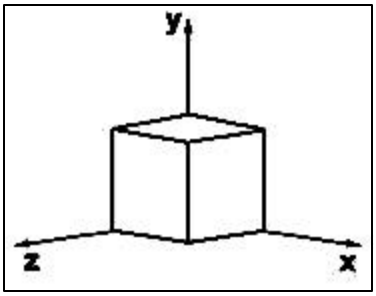


Figure 7 Dimetric Projection

Trimetric projections, the third subclass of axonometric projections, allow the viewer the most freedom in selecting the components of n (Hill, 1990). In this class, the viewplane normal makes differing angles with each major axis since no two components of **n** have the same value. As with a dimetric view, a trimetric view displays different orientations by placing differing amounts of emphasis on the faces. Trimetric projections have a potential disadvantage in that measurement of lines along the axes is difficult because of a difference in scaling factors. Figure 8, a trimetric view of a cube, shows how this unequal-foreshortening characteristic affects line measurements along different axes. While disadvantageous in maintaining measurements, a trimetric projection, with the correct orientation, can offer a realistic and natural view of an object (Hill, 1990).

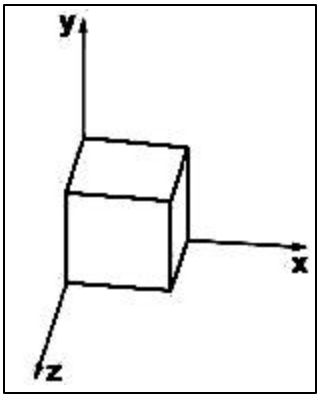


Figure 8 Trimetric Projection

Oblique Projections

Oblique projections represent the second category of parallel projections. Oblique views are useful since they combine the advantageous qualities of both multiview and axonometric projections. Like an axonometric view, this class presents an object's 3D appearance. Similar to a multiview projection, oblique views display the exact shape of one face (Hill, 1990). As in an orthographic view, this class of projections uses parallel projectors but the angle between the projectors and the viewplane is no longer orthogonal. Figure 9 shows an example of the direction of the projectors in relation to the viewplane.

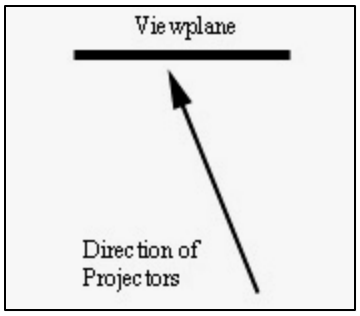


Figure 9 Direction of projectors for an oblique projection.

Oblique projections are further defined as either **cavalier** or **cabinet** projections. Figure 10 shows the projection of a point (x, y, z) to the point (xp, yp) onto the viewplane. Cavalier and cabinet projections differ by the value used for the angle *alpha.* Angle *alpha* is defined as the angle between the oblique projection line from (x, y, z) to (xp, yp) and the line on the viewplane from (x, y) to (xp, yp) (see Figure 10). Two commonly used values for *alpha* = 45°, and *alpha* = 63.4°.

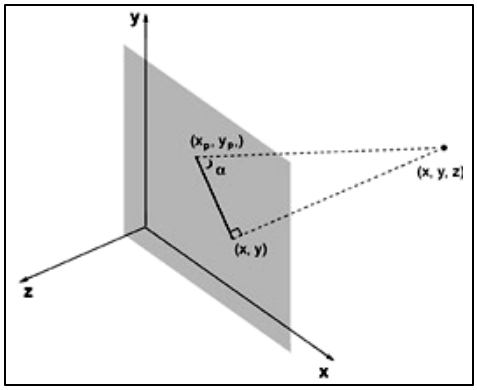
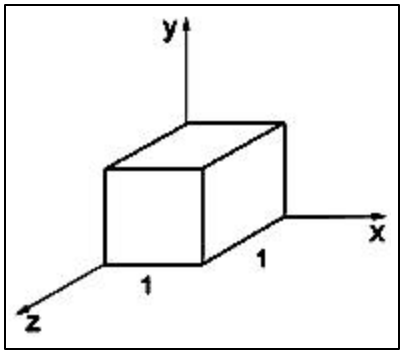


Figure 10 Conversion of a world coordinate point (x, y,z) to the position (xp,yp) on the viewplane for an oblique projection.

When *alpha* = 45° as in Figure 11, the projection is a **cavalier** projection. In this projection, a cube will be displayed with all sides maintaining equal lengths (see Figure 11). This property is often advantageous since edges can be measured directly. However, the cavalier projection can make an object look too elongated.

Figure 11 Cavalier Projection



In the second case, when *alpha* = 63.4°, the projection is labeled as a **cabinet** projection. For this angle, lines perpendicular to the viewplane are displayed one-half the actual length (see Figure 12). Because of the reduction of length in lines perpendicular to the viewplane, cabinet projections appear more realistic than cavalier projections (Hearn & Baker, 1996).

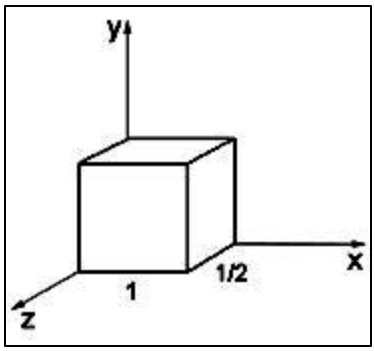


Figure 12 Cabinet Projection

Summary

We have seen a projections in this unit and its work what are the principle behind that. We have seen the parallel and perspective projections and their some of their different sub classifications like axonometric projections, oblique projections ,trimetric projections, isometric projections, multiview projections etc. we have seen the implementations of projection , hierarchy chart of different types of projection ,some examples to demonstrate all these terminology. However, in the next lesson, we will work on the geometric representations.

Questions

1. Explain 2D transformations with suitable example.
2. How to rotate a bit map.
3. How to display 24 bit image in 8 bit
4. How to detect corner in a collections of points.
5. Explain Scaling with suitable example.
6. Why Rotation is important?
7. Explain Affine transformation. Explain its difference with Shear transformation.
8. For 2D data, explain an economy way to transform this data to effect:
   A. Reflection about the Y-axis
   B. Reflection about a line through the origin inclined at an angle that to the Y-axis

   C. Reflection about a line parallel to the Y-axis passing through the point xo in the X-axis.
9. Explain window to viewport transformations.
10. What is matrix? Explain its representation on 3D.
11. How to generate a circle through three points.
12. How to find intersection of two 2D line segments.
13. Find a distance from a point to a line.
14. How to rotate a 3D point.
15. How to perform basic viewing in 3D.
16. How to optimize/simplify 3D transformation.
17. Give three points in 3D space: (x1, y1, z1), (x2.y2, z2) ,(x3,y3,z3)
    A. Derive an algebraic equation for the closest distance from the origin to the plane surface through these points
    B. Under what circumstances is this Zero
18. What is projection? Explain its various types.
19. Define perspective projection, its types and implementation
20. Explain parallel projection. What are its various types, explain in detail.
21. What do you mean, transformation can be about arbitrary access.
22. What do you mean, transformation can be centered on arbitrary points.
23. What are viewing and projection matrices.
24. Explain orthographic projection.
25. Explain Multiview projection.
26. Explain implementation of perspective projection
27. Write short notes on -
    - Axonometric Projections
    - Dimetric projections
    - Trimetric projections
    - Oblique Projections

Notes:

LESSON 22
HIDDEN LINE SURFACE REMOVAL METHOD

Topics Covered In The Unit

- Introduction to the geometric transformations
- What is Surface Removal method
- Z Buffer Algorithm
- Ray tracing
- Illumination and shading.
- Illumination models
- Shading models

Learning Objectives

Upon completion of this chapter, the student will be able to :

- Explain what is Surface Removal method .
- Z Buffer Algorithm
- Introduction to ray tracing
- Introduction to illumination and shading.
- Illumination models

Today's Topics

- Introduction to the geometric transformations
- What is Surface Removal method

Introduction

How to model 3D objects using polygons, and to compute realistic geometric transformations which realistically model the geometry of light interacting with our eyes. This enables us to display 3D wire-frame diagrams. How can we give a realistic look to 3D scenes? There are a variety of techniques available in computer graphics.

Of course, taking liberty with realistic effects can be a useful technique for conveying information (e.g., use of orthographic projection in computer-aided design). But we need to know the rules before we can decide when to bend them.

Techniques Include

- Visual surface determination - which objects in a scene are visible to the viewer. This is an extension of the geometric processing we have already studied.
- Shading - what color value to assign to each visible pixel. This is a complex topic that we will study extensively.
- Texture mapping - how to realistically model the details of textured and polychromatic surfaces.
- Dynamic modeling - to achieve realism in animated scenes.

Biggest problem: the complexity of the real world.

- Objects are illuminated not just by radiant light sources, but by light reflected from (or transmitted through) other objects.
- The surfaces of objects are not uniform in color and texture, but contain imperfections.
- Surface characteristics which determine how light interacts with an object are determined by molecular-level surface details.
- Projecting from 3D world to 2D display causes a loss of information. How can we make up for this loss?

We need to choose models and rendering techniques which will give the viewer enough information to be able to interpret images correctly, but are computationally efficient enough to be feasible on existing computer systems. (Note the tradeoff between realism and efficiency.)

What to do? Some techniques which may be applied to wire-frame drawings to achieve greater realism by giving more information to the viewer:

- Perspective projection. Problem: Objects that appear to be smaller, may actually *be* smaller, rather than farther away!
- Depth cueing. Make objects that are farther away dimmer. Exploits haze in the atmosphere, limits of focusing ability of our eyes. Implement intensity as a function of distance.
- Color. Assigning different colors to different objects or parts of objects can give important information in a wire-frame drawing.
- Hidden-line removal. Even if we are not filling polygons, we can achieve some positive effects through hidden line removal; that is, the removal from a drawing of line segments which are obscured by polygon faces nearer to the viewer.

If we fill the polygons in a scene with color, we can apply addtional techniques:

- Visible surface determination (i.e., hidden surface removal). Remove from a scene those objects and parts of objects which are obscured by other objects nearer to the viewer.
- Illumination and shading. An object of solid color appears to a viewer as a collection of shades, due to the effects of nonuniform lighting. Determining the correct shade of color for a pixel is a complex process, which depends on:
  - The position, orientation, brightness, and color of sources of illumination,
  - The characteristics of the material on the surface of the object, and
  - The position and orientation of the object in relation to the viewer and light sources.
- Interpolated shading. The assignment of a color to a pixel is performed by interpolating from polygon vertices. This is an important technique for giving a curved appearance to a polygonal object model. (Gourad and Phong shading.)

- Material properties. For a shading model to give a realistic appearance, it needs to allow for some variation in the material properties of the surfaces of objects. In particular, the shininess of a surface will affect the way that it reflects light, and therefore the way it appears to a viewer.
- Curved surface models. Polygonal mesh modeling is very powerful technique for approximating curved surfaces, especially when used with interpolated shading. Nevertheless, it is still an approximation. It may be possible to create more realistic images by directly representing curved surfaces with their true mathematical equations.
- Texture. The pattern in a textured object contains lines which, when viewed with a perspective projection, provide more depth information. In addition, texture mapping can be used to simulate the imperfections in a surface. Question: How could one use a torus to model a doughnut?
- Shadows. Shadows provide information regarding depth and relative position of objects, but are somewhat difficult to capture in a model. Standard shading methods are *local*, depending only on the object being viewed and the light sources in the scene. *Global* methods, in which the shade of an object depends also on the other objects in the scene, are much more computation intensive.
- Transparency and Refraction. Modeling of transparent or translucent objects requires additional work to achieve realism. As light passes through an object, it may refract (bend) or diffuse (as through fog). Modeling these effect properly requires the modeling of solid objects, not just surfaces.
- Improved camera models. Our camera model, based on a pinhole camera, has an infinite depth of field; that is, an infinite range of distances in which objects will appear in focus. We may be able to increase realism by simulating the lens in a camera to reduce this depth of field, making very near or far objects appear out of focus. This technique is used extensively by painters and photographers.

Some other modeling techniques include

- Improved object modeling. A variety of advanced mathematical techniques have been used to model objects. Fractals (e.g., the Koch curve) can be used to model objects with irregular surfaces. Particle systems can be used to model sets of particles which evolve over time. They have been successfully used to model fog, smoke, fire, fireworks, trees, and grass. Realistic depiction of some physical phenomena (ocean waves, draping of cloth, turbulence) can sometimes be modeled using differential equations from fluid mechanics or solid mechanics.
- Stereopsis. Present different images to the viewer's two eyes.

Hidden Surface Removal

(also known as Visible Surface Determination)

Basic problem: How do we handle the situation in which objects obscure other objects because they are nearer to the viewer; that is, the light rays from one object to the viewer are blocked by another object.

If we do nothing, and just draw primitives in random order, our screen image may show the wrong objects or parts of objects.

Assumptions

- We have a set of polygons to render.
- We have already performed geometric transformations we are ready to do the final projection and scan conversion.

There are a variety of algorithms, which vary in

- Resource (space and time) required
- How well (i.e., correctly) they work

We can classify algorithms into two categories:

Object space (Object precision). Compare polygons A and B pairwise. There are four cases to consider:

1. A completely obscures B. (Draw A, not B.)
2. B completely obscures A. (Draw B, not A.)
3. Neither one obscures the other. (Draw both.)
4. A and B partially obscure each other. (This is the hard part - it is necessary to calculate the visible parts of each.)

This approach leads to complexity of $O(k^2)$ where k is the number of polygons in the scene.

Image space (Image precision).

- For each pixel on the screen, cast a ray from the center (or direction) of projection through the pixel.
- Which polygon does the ray hit first? (i.e., which intersection point has the smallest z value in camera coordinates?)
- Color the pixel based on the properties of that polygon.

In this case, the complexity is O(kM), where k is the number of polygons in the scene and M is the number of pixels.

General Principles of Visible Surface Algorithms

- Exploit coherence.
- Perspective Transformation preserves *relative* depth.
- Extents and bounding volumes.
  - Object shapes are irregular, making it difficult to determine whether or not they intersect or obscure each other.
  - Simplify testing by enclosing each object in an *extent* or *bounding volume*.
  - The simplest form of bounding volume consists of x-, y-, and z-extents (between x_{min} and x_{max}, etc.)
  - Used to determine intersection of two objects, or between an object and a projector - if the extents don't intersect, then the objects don't, either.
- Backface Culling

- Polygon faces which are facing away from the viewer can be removed from a scene before further processing is done. Backfaces can be detected using the surface's normal vector **n** and a vector **v** pointing from the center of projection to a point on the surface. If **n** • **v** <= 0, it means that **n** and **v** are opposed to each other, so the surface is facing toward the viewer, and may be visible. If **n** • **v** > 0, the surface is a backface and cannot be visible.
- Removing backfaces reduces the work of whatever visible surface algorithm is being used.
- Backface culling is performed in OpenGL using the following functions:

glCullFace(mode); (mode = GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK)
glEnable(GL_CULL_FACE);

- Spatial partitioning is a technique used in many visual surface algorithms.
- Hierarchical models.

Shadows in OpenGL

The traditional approach to creating shadows is to apply the techniques that are used in visible surface determination. Visible-surface algorithms determine which surfaces can be seen from the viewpoint (the center of projection); shadow algorithms determine which surfaces can be "seen" from the light source. Thus, visible-surface algorithms and shadow algorithms are essentially the same.

- Determine which surfaces can be "seen" from a light source.
- At each pixel, combine the ambient term with diffuse and specular terms for all light sources visible from that pixel.

We'll come back to this after we study visual surface determination algorithms. In the meantime, we can see a shadow drawing method that can be used in OpenGL, for the special case of a shadow projected onto a planar surface, by making the following observation:

The shadow of an object on a surface is formed by projecting the object onto the plane of the surface, using the light source as the center of projection.

So, the shadow can be drawn with the following strategy:

- Treat the shadow as a separate object to be drawn. (The object actually is drawn twice: once to draw its own image, once to draw its shadow.)
- Draw the scene in this order: First the surface on which the shadow will appear is drawn normally. Then the shadow object is drawn, using only the ambient light component. Then the object is drawn normally.
- The shadow object is drawn using a manipulation of the modelview matrix, which computes the projection of the object onto the surface. (The projection matrix is used to project the shadow onto the view plane in the usual way. So, the object is actually projected twice: once onto the surface using the light source as center of projection, then onto the view plane using the camera position as center of projection.)
- If the object consists of a set of polygons (e.g. a polygonal mesh), the above strategy is applied to every polygon in the set.

To use the modelview matrix to apply the shadow projection, the following transformations are applied:

- Translate the light source to the origin (T(-lx,-ly,-lz)).
- Project onto the surface. To project a point onto a plane with equation Ax + By + Cz + D = 0, using the origin as center of projection, use the matrix

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -A/D & -B/D & -C/D & 0 \end{bmatrix}$$

- Then translate back to the original position of the light source (T(lx,ly,lz)).
- Finally, the object can be drawn, using whatever modeling transformations are used to model it.

Notes:

LESSON 23
Z BUFFER ALGORITHM

Today's Topics

· **Z Buffer Algorithm**

The z-buffer Algorithm

- Most commonly used
- Easy to implement in hardware or software
- Works well in pipelined graphics architectures
- OpenGL uses it

Uses the *z-buffer* (also called the *depth buffer*), a memory area consisting of one entry per pixel. The z-buffer contains the z (depth) value for the point in the scene which is drawn at that pixel. Assume that the negative z-axis points away from the viewer, so that the smaller the z value, the farther away from the viewer.

The algorithm in pseudocode: (zB is the name of the z buffer, frameB is the name of the frame buffer.)

```
Set all z-buffer values to zMIN;
Set all pixels to the background color;
For each polygon in the scene {
      Perform scan conversion;
      For each point P(x,y,z) do {
            if z >= zB[x][y]{ // if the point is closer to the
viewer than the point already drawn at that pixel
                  frameB[x][y] = P.color; // draw the point at the
pixel
                  zB[x][y]=z; // update the z buffer
            }
            if( z < zB[x][y] ) // if the point is farther from the
viewer than the point already drawn at that pixel
                  do nothing;
      }
}
```

- Polygons can be drawn in any order (no presorting necessary).
- No object-object comparisons (an image space algorithm)
- Works for curved surfaces as well as polygons
- Space requirement
- Subject to aliasing, due to limited precision of the z-buffer.

Z-Buffering, or, My Z's Bigger Than Yours, a little more on z buffer

Among the competing consumer 3D architectures-GeForce 2/3, ATI's Radeon and STMicro's Kyro II-there's a considerable variation in alternate z-buffering methods. These adhere to the 3D graphics mantra of "don't do what you don't have to", and all endeavor to avoid unneeded rendering work by doing z-buffering tests earlier before other tasks get performed, most notably texture mapping. Appreciate that z-buffering isn't the only method to determine visibility.

A fairly rudimentary z-buffering technique system is called the Painter's Algorithm (a back-to-front method), which begins from the back of the scene and draws everything in the scene, including full rendering of objects that might be occluded by objects nearer to the camera. In most cases, this algorithm achieves correct depth sorting, but it's inefficient, and can cause some drawing errors where triangles overlap one another. Further, it is costly in terms how many times a given pixel might be rendered.

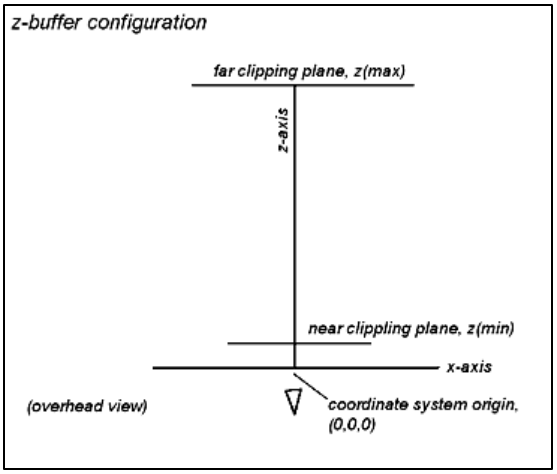
Back in the days where 3D engines were run entirely on CPUs, the Painter's algorithm was used because a z-buffer wasn't readily available. But this algorithm is hardly used anymore as all consumer 3D graphics hardware now has z-buffering as a standard feature.

The multiple renders of a particular pixel in a scene is called overdraw. Averaged across the scene, this overdraw is often termed the scene's "depth complexity" and describes how many "layers" of objects you're looking at on average in a scene. Most games currently have a depth complexity of around 2.5 to 3.

Remember that although screen space is pretty much a "2D" mapping to the screen coordinates, say 1600x1200, the z-values have been carried through all of the previous operations in the pipeline.

But with all modern hardware having a z-buffer, this is now the preferred method of depth testing, as it is more efficient, and produces more correct results. To set the z-buffer setup for depth tests for the next frame of animation, (generally at the end of the previous scene being drawn) the z-buffer is cleared, meaning that the value of z_{max} gets written to all pixel positions in preparation for the next frame of animation's depth testing.

Here's another example where multiple methods exist to perform the same operation. An application can set up its z-buffer such that positive z-axis values go away from the view camera, or that negative z values go away from the view camera. See the Z-Buffer diagram.



For the sake of our example, let's set z_{min} at the near clipping plane, with psitive z going away from the view camera. Irrespective of which way the z-buffer gets configured, depth testing is a pixel-by-pixel logical test that asks: "is there anything in front of this pixel?" If the answer is returned yes, that pixel gets discarded, if the answer is no, that pixel color gets written into the color buffer (back buffer) and the z-buffer's z-value at that pixel location is updated.

Another choice in how depth buffering gets done is "sort ordering", with the choices being front-to-back or back-to-front. But, z-buffering can do its depth tests either way, though back-to-front seems to be the preferred method. Sounds simple enough, right?

Well, another maxim of 3D graphics is that things are rarely as simple as they seem. Here are some the potential perils of z-buffering:

1. First, a 16-bit z-buffer has 65,535 different values (0 to 65,534), which one would think of as being plenty of accuracy. But there are two problems, one having to do with the scale of the 3D scene, and the other having to do with the non-linear behavior of values in the z-buffer. If an application is drawing a large, outdoor environment, say in a flight simulator, where the viewer may be looking at tens or hundreds of miles of terrain, a 16-bit z-buffer may not provide sufficient resolution for all objects in the scene.
2. Secondly, the z-values in screen space don't behave in a linear fashion, because they, like the x and y values back in clip space, were divided by w during the perspective divide. This non-linearity is a result of using a perspective projection (which creates the view frustum). The result is much greater accuracy or sensitivity for z-values in the near field, and then values become less accurate at points further away from z_{min}.

One way programmers can gain more consistent accuracy using the z-buffer is to set the near clipping plane further out, and bring the far clipping plane closer in, which brings the ratio of z-near to z-far closer to one, and evens out the variance in accuracy. [RTR, pp. 368-9] The trick of "compressing" the z-near and z-far planes is often used in conjunction with CAD programs, and is one way to even out the z-buffer's accuracy.

Another approach to solving the issue of uneven accuracy in depth values is to use w in lieu of z for depth buffering. Recall that the w is a scaled view-space depth value, and rather than write the z/w value for each vertex, the w value itself is instead used for depth tests. In some sense, w-buffering occurs whether it's being used for depth buffering or not, since a value of 1/w is used to interpolate pixels across triangle spans (the "space" between the two points where a triangle crosses a given scan-line) to produce perspective-correct texture mapping. [RTR, p. 369] But additionally, w can be used for depth buffering, and w values, which are floating-point values between 0 and 1, produce a linear scale from z-near to z-far, providing consistent accuracy.

According to Jim Blinn, in *Jim Blinn's Corner: A Trip Down the Graphics Pipeline*, (Morgan Kaufman, San Francisco, 1996), the decision to use w or z for depth buffering is best determined based on the ratio of z-near to z-far (z_{min} to z_{max}). His conclusions are:

- If z-near/z-far = 0.3, use w-buffering
- If z-near/z-far > 0.3, probably use z-buffering
- And if z-near/z-far is > 0.5, definitely use z-buffering [BLINN]

After the depth buffering tests have completed, and the front and back buffers get page-flipped, the z-buffer is "cleared", where the z_{max} value gets written into each pixel position in preparation for the next round of depth buffering tests.

One curious thing about the way traditional depth buffering has been done is the fact that is done so late in the pipeline, after texturing, fog and alpha operations have been processed. In other words, a good number of pixels will be thrown out at the last minute after having been textured, alpha-tested/blended and had fog applied. For this reason, ATI, nVidia and STMicro all have alternative approaches that seek to either move depth buffering further up the pipeline, or make depth buffering operations themselves faster.

Notes:

LESSON 24
RAY TRACING

Today's Topics

· **Ray Tracing**

Introduction to visible Surface Ray Tracing

- Ray tracing determines the visibility of surfaces by tracing imaginary rays of light from the viewer's eye to the objects in the scene.
- A center of projection (the viewer's eye) and a window on an arbitrary view plane are selected. The window may be thought of as being divided into a regular grid whose elements correspond to pixels at the desired resolution.
- For each pixel in the window, an eye ray is fired from the center of projection through the pixel's center into the scene.
- Computing the intersections of the ray with the objects is at the heart of any ray tracer. It's easy for planes, spheres, more difficult for other objects.
- We must also determine the surface normal at the point of intersection in order to shade the surface

Simple ray tracing algorithm:

```
for (each scan line in image) {
   for (each pixel in scan line ) {
      determine ray from eye through the center of the pixel;
      for(each object in scene) {
         determine the intersection between the ray and the
object;
         if(object is intersected and is closest considered thus
far)
            record intersection point and object id.
         }
      determine the surface normal at the closest intersection
point;
      apply the illumination model to the intersection point,
and use the result to color the pixel;
      }
   }
```

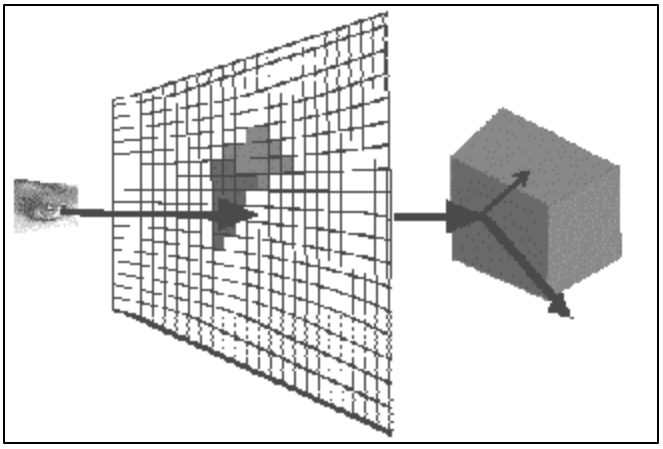
Problem: The number of intersections to be computed is k * M, where k is the number of objects in the scene and M is the number of pixels. For a scene of 100 polygons displayed on a 1024x1024 viewing window, this is 100 million intersections.

Some techniques for improving efficiency:

- Optimize intersection calculations.
  - Precompute parts of the intersection formula which are ray-independent or object-independent.
  - Apply coordinate transformation to simplify calculations.
  - Intersect with bounding volume first (for complex objects)
- Organize bounding volumes in hierarchies.
  - Create a super-bounding volume which encloses a set of bounding volumes.
  - Continue until a tree structure is created.
  - To find a given ray's closest intersection with an object, traverse the tree, beginning with the root.
  - If a ray does not intersect a parent bounding volume, it is not necessary to consider any of its children.
- Spatial partioning.
  - Conceptually similar to BSP tree - construct a partitioning of space, top-down.
  - Start with a bounding box for the entire scene.
  - Recursively split the box into subboxes, keeping a list of the objects which are completely or partially contained in each.
  - When processing a ray, it is necessary to find the intersections only of the objects contained in partitions intersected by the ray.

Antialiasing can be done by casting (adaptively) several rays per pixel.

A Little More Detail



The diagram illustrates a partially complete ray tracing. We see on the (oblique) screen a partial image of the cube data object. The turquoise pixel is just about to be rendered. Let's "listen in" on what happens next...

- The program begins by "shooting" a ray from the hypothetical eye of the observer, through the pixel in question, and into "the data".
- The ray is tested against all the polygons of the model to see if it intersects any. If it does not, the pixel is colored the background color.
- If the ray intersects one or more polygons, the one nearest the screen is selected. The ray's angle of incidence is

calculated, and the surface's index of refraction is looked up.

- Now, **TWO** rays are created leaving the point of intersection. One is the **reflected ray** and the other is the **refracted ray**.
- If we are calculating shadows, a ray is shot towards each light source. It will test to uncover shadowing objects.
- For EACH active ray, return to the second step and start testing again. Stop the process after a certain number of loops, or when a ray strikes the background.

This process can produce a very large number of rays to test, all to establish the color of just one pixel. No wonder it takes so long.

Another reason it takes a long time is that there is little "coherence" to the process. That is, the algorithm can't use much information from the adjacent pixels. It pretty much does each pixel independently.

Much research has been done on ways to improve ray tracing speed, with lots of concentration on the "ray-polygon intersection test", which is where lots of computation time is spent. Different strategies, which assign polygons to subregions of space, make it possible to eliminate whole groups based on the general direction in which the ray is going.

Another unfortunate quality of ray tracings is that **NONE** of the calculation time helps you when you change the view point slightly and start rendering a new frame, as in an animation. Again, you start over.

Notes:

LESSON 25
INTRODUCTION TO ILLUMINATION AND SHADING AND ILLUMINATION MODELS

Today's Topics

- **Introduction to Illumination and Shading**
- **Illumination models**

Introduction

We have seen how to model 3D objects , and to compute realistic geometric transformations which realistically model the geometry of light interacting with our eyes. This enables us to display 3D wire-frame diagrams. To achieve greater realism, we need to color (fill) the objects in a scene with appropriate colors.

One basic problem: Even though an object is made from a single, homogeneous material, it will appear to the human eye as being painted with different colors or shades, depending on its position and orientation, the characteristics of the material of which it is made, and the light sources which are illuminating it. If we fill an object with a solid color, we will actually provide the viewer with less information than a wire-frame drawing would. So, in order to color objects realistically, we need to color them as them human eye sees them. This requires models to represent light sources, surface characteristics, and the reflection of light from surfaces to the eye of the viewer.

We are interested in:

- Illumination models, which describe how light interacts with objects, and
- Shading models, which describe how an illumination model is applied to an object representation for viewing.

Models which would completely capture the physics of light and how it interacts with surfaces and the vision system would be too complex to implement. (Why?). Instead, the models which are used have little grounding in theory, but work well in practice. Problem: A model may work well in capturing one effect, but not so well with others. Models may vary widely in the amount of computation time required.

Illumination Models

The light which illuminates a scene is comprised of direct lighting from radiant sources and light reflected from nonradiant objects. The reflected light is modeled as *ambient light* . Both radiant light and ambient light are reflected from object surfaces to the eye of the viewer.

Ambient light

Ambient light consists of multiple rays of light reflected from the many surfaces present in the environment. We assume that ambient light impinges equally on all surfaces from all directions.

How does ambient light contribute to the view of an object by a viewer? The intensity of the ambient light which is reflected by an object into the eye of the viewer is modeled by the equation

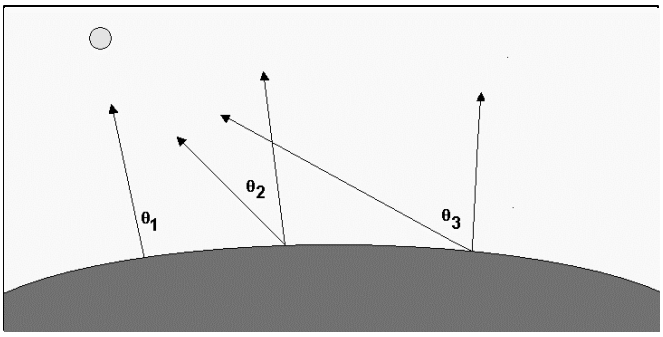
$I = I_a K_a$

where I_a is the intensity of the ambient light and K_a is the *ambient-reflection coefficient* . K_a ranges from 0 to 1, and is a material property of the object's surface. (The computed I value determines the intensity of the object on the screen.)

Next, consider a radiant light source. The simplest way to model such a source is as a single point, which we call a *point source* . A point source reflecting from a surface is modeled as *diffuse reflection* and *specular reflection* .

Diffuse Reflection (Lambertian Reflection)

Dull, matte surfaces exhibit diffuse reflection. These surfaces appear equally bright from all viewing angles because they reflect light with equal intensity in all directions. However, the brightness does depend on the angle at which the light source hits the surface; the more direct the light, the brighter the surface.



$I = I_p K_d \cos(\text{theta})$

where I_p is the point light source's intensity, K_d is the material's diffuse-reflection coefficient, and theta is the angle of incidence between the light source and the material surface normal. (What is the rationale for this equation?)

K_d is a constant from 0 to 1 which depends on the material of the surface. The angle theta is between 0 and 90 degrees. If the surface normal and light direction vectors are normalized, this can be replaced by

$I = I_p K_d (N \cdot L)$

If a point light source is sufficiently distant from the objects being shaded (e.g., the sun), it makes essentially the same angle with all surfaces sharing the same surface normal.

An object illuminated by a single point source looks like an object in the dark lighted by a flashlight. An object in daylight has both an ambient component and a diffuse reflection component, giving:

$I = I_a K_a + I_p K_d (N \cdot L)$

LESSON 26
ILLUMINATION MODELS (CONTD…)

Today's Topics

- **Illumination Models Contd.**

Light-source Attenuation

The energy from a point light source that reaches a given part of a surface falls off as the inverse square of the distance from the source to the surface. If we add this effect to our model, we get

$I = I_a K_a + f_{att} I_p K_d (N \cdot L)$

where $f_{att} = 1/d^2$

In practice, however, this often does not work well. If the light is very close, it varies widely, giving considerably different shades to surface with the same angle theta between N and L.

An empirically useful compromise for the attenuation factor is:

$f_{att} = \min (1, 1/(c_1 + c_2 d + c_3 d^2))$

Here c_1, c_2, c_3 are constants. c_1 keeps the denominator from becoming too small.

Colored lights and Surfaces

Colored illumination can be represented by describing incident light in terms of three color components (e.g. RGB).

An object's color can then be specified by giving three reflectivities, one for each color component, and writing three equations.

Not completely correct, but often produces reasonable results.

The triple (O_{dR}, O_{dG}, O_{dB}) defines an object's diffuse red, green, and blue components in the RGB color system.

In this case, the illuminating light's three primary components, I_{pR} , I_{pG} , and I_{pB} , are reflected in proportion to O_{dR}, O_{dG}, and O_{dB} , respectively. For example, the red component is computed by:

$I_R = I_{aR} K_a O_{dR} + f_{att} I_{pR} K_d O_{dR} (N \cdot L)$

Atmospheric Attenuation

Depth cueing - more distant objects are rendered with lower intensity than are objects closer to the viewer. (Note: This is not the same as light source attenuation.) How to model it?

For each primary color, we have

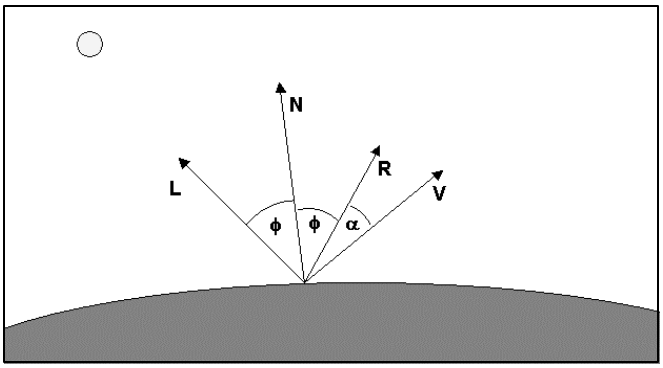
$I'_\lambda = S_0 I_\lambda + (1 - S_0) I_{dc\lambda}$

where S is the scale factor for atmospheric attenuation; $I_{dc\lambda}$ is defined by depth-cue color which allows shift in color caused by the intervening atmosphere. "

Specular Reflection

Illuminate an apple with a bright white light: the highlight is caused by *specular reflection*. At the highlight, the apple appears not to be red, but white, the color of the incident light. Such highlights are produced by non-uniform reflection from shiny surfaces. A perfectly specular surface is like a mirror, reflecting a point source as a point.

The more typical situation is partial specularity where the specular reflection occurs primarily (but not exclusively) in the direction of perfect reflection, and falls off rapidly away from this direction. The amount of specular reflection seen by a viewer depends on the angle between the viewing direction V and the direction of perfect reflection R.



The Phong illumination model approximates this effect with a term of the form $W(\theta) \cos^n \alpha$ where α is the angle between the viewing direction and the direction of perfect specular reflection. n is typically somewhere between 1 and a few hundred.

$W(\theta)$ is often set to a constant, the specular reflection coefficient.

The viewer can see specularly reflected light from a mirror only when the angle a is zero; a is the angle between R and the direction of the view point V

The Phong Illumination model

The Phong model combines the effects of ambient light, diffuse reflection, and specular reflection to determine the the intensity of light at a point.

Specular reflection is affected by the properties of the surface

Improvements to the lighting model

So far, we have modeled light sources as points radiating uniformly in all directions. The Warn model improves this by letting a light source be aimed in a certain direction. The intensity of light then varies with the direction from the source.

In the Warn model, this directionality is given by $\cos^n \theta$ where θ is the angle from the central (most intense) direction. The larger the value of n, the more concentrated the light is in a certain direction.

Can implement "shutters" to restrict light to a certain region of space.

Multiple point sources can also be used - just sum their effects. This leads to an extension of the Phong model: Extended sources are light sources with area. They require more complex methods.

LESSON 27
SHADING MODELS

Today's Topics

· **Shading models**

Shading Models

Shading models determine how a given intensity of incident light is to be used to illuminate a point, i.e., a pixel. We assume here that an object is modeled by a polygonal mesh, which may be only an approximation to the object itself. A normal vector, which is necessary for our lighting model, can be computed for each polygon in the mesh.

Constant Shading

- Also known as flat shading.
- Apply the illumination model once per polygon to obtain color for whole region.
- Gives a faceted effect.
- Valid if the modeled objects are actually polyhedral, and the light source and viewer are at infinity.
- Undesirable if polygons are being used to model a curved surface.
- Increasing the number of facets is not as effective as might be thought because of Mach banding, which exaggerates the perceived intensity change at any edge where there is a discontinuity in magnitude or slope of intensity. At the border between two facets, the dark facet looks darker and the light facet looks lighter.

When a polygon mesh is used to model a curved surface, it is better to use some form of *interpolated shading*. Gourad shading and Phong shading are two forms of interpolated shading.

Gouraud Shading

- Also called smooth shading, intensity interpolation shading or color interpolation shading, it is used to eliminate intensity discontinuities.
- How it works:
  - Approximate the normal at each vertex by averaging the normals of the polygons sharing the vertex.
  - Find the vertex intensities by using the vertex normals with any desired illumination model.
  - Linearly interpolate intensity values along edges.
  - Linearly interpolate intensity values along scan lines between edges.
- The interpolation along edges can easily be integrated with the scan-line visible-surface algorithm.
- Use multiple normals if we want edge to be visible.
- Faceted appearance greatly reduced, but not always completely eliminated, especially in regions with strong contrasts.
- Highlights sharper than individual polygons don't work well.

Phong Shading

- Also known as normal-vector interpolation shading, it interpolates the surface normals instead of the intensity values.
- A normal vector is computed by linear interpolation for each point in the polygon.
- The illumination model is applied to each point, using the interpolated normal, during scan conversion.
- Computationally more expensive.
- Gives better results, especially for highlights.

If Gouraud shading is used, then the intensity across the polygon is linearly interpolated. If a highlight fails to fall at a vertex, then Gouraud shading may miss it entirely. In contrast, Phong shading allows highlights to be located in a polygon's interior.

Notes:

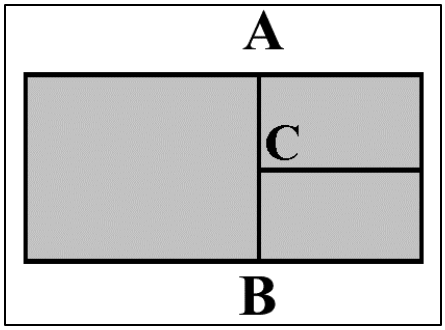
LESSON 28
SHADING MODELS (CONTD...)

Today's Topics

· **Shading Models Contd.**

Problems with Interpolated Shading

- Polygonal silhouette. No matter how good an approximation an interpolated shading model offers to a curved surface, the silhouette edge of the mesh is still clearly polygonal.
- Perspective distortion may occur, since interpolation is performed on the projected image.
- Orientation dependence (solve by using triangles).
- Problems if two adjacent polygons fail to share a vertex (see figure below).



- Insufficient sampling can produce unrepresentative vertex normals (essentially a form of aliasing).
- Although these problems have prompted much work on rendering algorithms that handle curved surfaces directly, polygons are sufficiently faster to process that they still form the core of most rendering systems.

The "standard" reflection model in computer graphics that compromises between acceptable results and processing cost is the Phong model. The Phong model describes the interaction of light with a surface, in terms of the properties of the surface and the nature of the incident light. The reflection model is the basic factor in the look of a three dimensional shaded object. It enables a two dimensional screen projection of an object to look real. The Phong model reflected light in terms of a diffuse and specular component together with an ambient term. The intensity of a point on a surface is taken to be the linear combination of these three components.

A. Diffuse Reflection

Most objects we see around us do not emit light of their own. Rather they absorb daylight, or light emitted from an artificial source, and reflect part of it. The reflection is due to molecular interaction between the incident light and the surface material. A surface reflects coloured light when illuminated by white light and the coloured reflected light is due to diffuse reflection. A surface that is a perfect diffuser scatters light equally in all directions. This means that the amount of reflected light seen by the viewer does not depend on the viewer's position. The intensity of diffused light is given by Lambert's Law:

Id = IiKdcosA (1.1)

Ii is the intensity of the light source. A is the angle between the surface normal and a line from the surface point to the light source. The angle varies between 0 and 90 degrees. Kd is a constant between 0 and 1, which is an approximation to the diffuse reflectivity which depends on the nature of the material and the wavelenght of the incident light. Equation 1.1 can be written as the dot product of two unit vector:

Id = IiKd(L.N) (1.2)

where N is the surface normal and L is the direction of vector from the light source to the point on the surface. If there is more than one light source then:

$$I_d = K_d \sum_n I_{in}(L_n \bullet N) \qquad \textbf{(1.3)}$$

B. Ambient Light

Ambient light is the result of multiple reflections from walls and objects, and is incident on a surface from all directions. It is modelled as a constant term for a particular object using a constant ambient reflection coeffient:

I = IaKa (1.4)

Where **Ia** is the intensity of the ambient light and **Ka** is the ambient reflection coefficient. Ambient light originates from the interaction of diffuse reflection from all the surfaces in the scene.

C. Specular Reflection

Most surfaces in real life are not perfectly diffusers of the light and usually have some degree of glossiness. Light reflected from a glossy surfac e tends to leave the surface along vector **R** , where **R** is such that incident angle is equal to reflection angle. The degree of specular reflection seen by the viewer depends on the viewing direction. For a perfect glossy surface, all the light is reflected along the mirror direction. The area over which specular reflection is seen is commonly referred to as a **highlight** and this is an important aspect of Phong Shading: the color of the specularly reflected light is different from that of the diffuse reflected light. In simple models of specular reflection the specular component is assumed to be the color of the light source. The linear combination of the above three components: diffuse, ambient and specular is as follows:

$$I = I_a K_a + I_i \left[K_d (L \cdot N) + K_s \cos^n B \right]$$

That is :

$$I = I_a K_a + I_i \left[K_d (L \cdot N) + K_s (R \cdot V)^n \right] \quad (1.5)$$

Where **B** is the angle between the viewing vector **V** and the reflection vector **R** . **Ks** is the specular reflection coefficient, usually taken to be a material-dependent constant. For a perfect reflector **n** is infinite. A very glossy surface produces a small highlight area and n is large.

D. Geometric Consideration

The expense of Equation 1.5 can be considerably reduced by making some geometric assumptions and approximations. If the light source and viewpoint are considered to be at infinity then **L** and **V** are constant over the domain of the scene. The vector **R** is expensive to calculate, it is better to use **H**. The specular term then becomes a function of **N.H** rather than **R.V**. **H** is the unit normal to a hypothetical surface that is oriented in a direction halfway between the light direction vector **L** and the viewing vector **V**:

$$H = (L + V) / 2 \quad (1.6)$$

The equation 1.5 becomes:

$$I = I_a K_a + I_i \left[K_d (L \cdot N) + K_s (N \cdot H)^n \right] \quad (1.7)$$

In this project, I would like to make the light postion changable and show the effect of shading for the different light position, so this assumption has not been used.

Notes:

LESSON 29
SHADING MODELS (CONTD...)

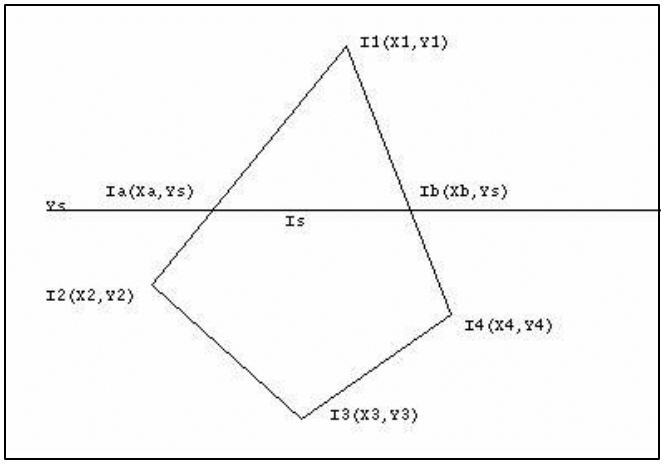
Today's Topics

· **Shading Models Contd.**

Implementation

I apply the above Phong reflection model to a dodecahedron model to show the advantage and disadvantage of Phong Shading and Gouraud Shading.

A. Gouraud Shading



(2.1)

In Gouraud Shading, the intensity at each vertex of the polygon is first calculated by applying equation 1.7. The normal **N** used in this equation is the vertex normal which is calculated as the average of the normals of the polygons that share the vertex. This is an important feature of the Gouraud Shading and the vertex normal is an approximation to the true normal of the surface at that point. The intensities at the edge of each scan line are calculated from the vertex intensities and the intensities along a scan line from these. The interpolation equations are as follows:

$$I_a = \frac{1}{y_1 - y_2}[I_1(y_s - y_2) + I_2(y_1 - y_s)]$$

$$I_b = \frac{1}{y_1 - y_4}[I_1(y_s - y_4) + I_4(y_1 - y_s)]$$

$$I_s = \frac{1}{x_b - x_a}[I_a(x_b - x_s) + I_b(x_s - x_a)]$$

(2.2)

For computational efficiency these equations are often implemented as incremental calculations. The intensity of one pixel can be calculated from the previous pixel according to the increment of intensity:

$$\Delta I_s = \frac{\Delta x}{x_b - x_a}(I_b - I_a)$$

$$I_{s,n} = I_{s,n-1} + \Delta I_s$$

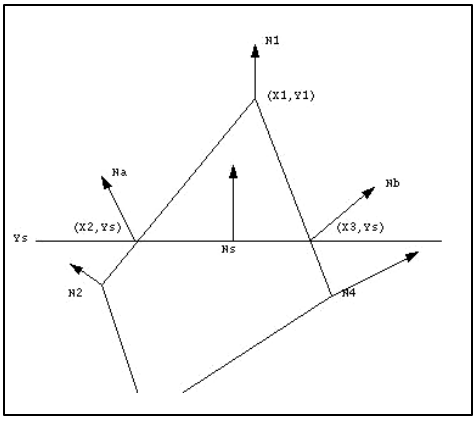
(2.3)

The inplementation of the Gouraud Shading is as follows:

```
deltaI = (i2 - i1) / (x2 - x1);
  for (xx = x1; xx < x2; xx++)
            {       int offset = row * CScene.screenW + xx;
                    if (z < CScene.zBuf[offset])
                    {  CScene.zBuf[offset] = z;
                       CScene.frameBuf[offset] = i1;
                    }
                    z += deltaZ;        i1 += deltaI;
            }
```

Where **CScene.ZBuf** is the data structure to store the depth of the pixel for hidden-surface removal (I will discuss this later). And **CScene.frameBuf** is the buffer to store the pixle value. The above code is the implementation for one active scan line. In Gouraud Shading anomalies can appear in animated sequences because the intensity interpolation is carried out in screen coordinates from vertex normals calculated in world coordinate. No highlight is smaller than a polygon.

B. Phong Shading



Phong Shading overcomes some of the disadvantages of Gouraud Shading and specular reflection can be successfully incorporated in the scheme. The first stage in the process is the same as for the Gouraud Shading - for any polygon we evaluate the vertex normals. For each scan line in the polygon we evaluate by linear intrepolation the normal vectors at the end of each line. These two vectors **Na** and **Nb** are then used to interpolate **Ns**. we thus derive a normal vector for each point or pixel on the polygon that is an approximation to the real normal on the curved surface approximated by the polygon. **Ns** , the interpolated normal vector, is then used in the intensity calculation. The vector interpolation tends to restore the

curvature of the original surface that has been approximated by a polygon mesh. We have :

$$N_a = \frac{1}{y_1 - y_2}[N_1(y_s - y_2) + N_2(y_1 - y_s)]$$

$$N_b = \frac{1}{y_1 - y_4}[N_1(y_s - y_4) + N_4(y_1 - y_s)]$$

$$N_s = \frac{1}{x_b - x_a}[N_a(x_b - x_s) + N_b(x_s - x_a)] \quad (2.5)$$

These are vector equations that would each be implemented as a set of three equations, one for each of the components of the vectors in world space. This makes the Phong Shading interpolation phase three times as expensive as Gouraud Shading. In addition there is an application of the Phong model intensity equation at every pixel. The incremental computation is also used for the intensity interpolation:

$$N_{sx,n} = N_{sx,n-1} + \Delta N_{sx}$$

$$N_{sy,n} = N_{sy,n-1} + \Delta N_{sy}$$

$$N_{sz,n} = N_{sz,n-1} + \Delta N_{sz} \quad (2.6)$$

The implementation of Phong Shading is as follows:

```
for (xx = x1; xx < x2; xx++)
        {       int offset = row * CScene.screenW + xx;
                if (z < CScene.zBuf[offset])
                {   CScene.zBuf[offset] = z;
                   pt = face.findPtInWC(u,v);
                   float Ival = face.ptIntensity;
                   CScene.frameBuf[offset] = Ival;< BR>
                }
                u += deltaU;
                z += deltaZ;
                p1.add(deltaPt);
                n1.add(deltaN);
    } M
```

So in Phong Shading the attribute interpolated are the vertex normals, rather than vertex intensities. Interpolation of normal allows highlights smaller than a polygon.

Summary

We have seen different geometric representations in this unit. We start with the Surface removal method also known as visible surface determination and its principle. We seen the Z buffer algorithm its algorithm in codes with a diagrammatic view. We move to introduction to ray tracing in the next heading seen its simple algorithm. next topic Illumination and shading we start with the introduction and move to is models in which Illumination models describe how light interacts with objects some of that are Ambient light , Diffuse Reflection (Lambertian Reflection),Light source attenuation, Phong illumination model etc and in Shading models describe how an illumination model is applied to an object representation for viewing some of the models are Constant shading ,Gouraud Shading ,phong Shading ,we see the phong and Gouraud shading model in detail with its geometric considerations and implementations , some examples to demonstrate all these terminology. However, in the next lesson, we will work on the Curves and Models..

Questions

1. Describe a hidden-line algorithm, and explain its advantages and disadvantages over other algorithms for the same purpose.
2. Describe briefly four different hidden line algorithms
3. How do a hidden surface test (backface culling) with 2D points?
4. Explain general principles of visible surface algorithms?
5. Explain z-buffer algorithm? Write pseudocode into algorithm?
6. What is painter's algorithm.?
7. What is ray tracing?
8. Explain visible ray tracing algorithm?
9. What is illumination? List various types?
10. Explain Ambient light illumination model?
11. Explain diffuse reflection illumination model?
12. Explain Phong illumination model?
13. Write short notes on: -
    a. Light-source attenuation
    b. Colored lights and surfaces
    c Atmospheric Attenuation
    d. Specular Reflection
14. What is shading?
15. Explain Constant Shading model?
16. Explain Gouraud Shading model?
17. Explain Phong Shading model?
18. Explain Interpolated Shading model?
19. Explain implementation of Phong Shading model?
20. Explain implementation of Gouraud Shading model?

LESSON 30
POLYGON MESHES

Topics Covered in the Unit

- **Introduction to Curves and Surfaces**
- **Polygon meshes**
- **Parametric cubic curves**
- **Hermite curves**
- **Beizer curves**
- **Introduction to Animation**
- **Key Frame Animation**

Learning Objectives

Upon completion of this chapter, the student will be able to :

- Explain what is Curves and Surfaces
- Explain Polygon meshes
- Explain Parametric cubic curves
- Explain Hermite curves
- Explain Beizer curves
- Explain what is Animation and the some of the designing tricks
- Explain what is Key Frame Animation

Today's Topics

- **Introduction to Curves and Surfaces**
- **Polygon Meshes**

Introduction

A few years back, consumer-level 3D acceleration was unheard of. Now, even the previously non-accelerated graphics machines on the low end have slowly been replaced with computers that can run 3D graphics using an accelerated chipset. However, graphics adapter manufacturers continue to create faster and faster chipsets, making products from 6 months ago seem slow. In order to support high-end consumers, and at the same time not abandon the low-end consumers, a new trend has been born in the industry: scalable geometry. Scalable geometry is any kind of geometry that can be adapted to run with decreased visual quality on slower machines or with increased visual quality on faster machines.

Curved surfaces are one of the most popular ways of implementing scalable geometry. Games applying curved surfaces look fantastic. UNREAL's characters looked smooth whether they are a hundred yards away, or coming down on top of you. QUAKE 3: ARENA screen shots show organic levels with stunning smooth, curved walls and tubes. There are a number of benefits to using curved surfaces. Implementations can be very fast, and the space required to store the curved surfaces is generally much smaller than the space required to store either a number of LOD models or a very high detail model.

The industry demands tools that can make creation and manipulation of curves more intuitive

Polygon Meshes

A *polygon mesh* is a collection of edges, vertices, and polygons connected such that each edge is sbared by at most two polygons. An edge eonnectsJwo vertices, and a polygon is a closed sequence of edges. An edge can be shared by two adjacent polygons, and a vertex is shared by at lea.<.t two edges. A polygon mesh can be represented in several different ways, each with its advantages and disadvantages. The application programmer's task is to choose the most appropiate representation. Several representations can be used in a single application: one for external storage, another for internal use, and yet<another with which the user interactively creates the mesh.

Two basic criteria, space and time, can be used to evaluate different representations. Typical operations on a polygon mesh are finding aU the edges incident to a vertex, finding the polygons sharing an edge or a vertex, finding the vertices connected by an edge, finding the edges of a polygon, displaying the mesh, and identifying errors in representation (e.g., a missing edge, vertex, or polygon). In general, the more explicitly the relations among polygons_ vertices, and edges are represented, the faster the operations are and the more space the representation requires. Woo [WOO85] has.analyzed the time complexity of nine basic access operations and nine basic update operations on a polygon-mesh data structure.

In the rest of this section, several issues concerning polygon meshes are discussed: representing polygon meshes, ensuring that a given representation is correct, and calculating the coefficients of the plane of a polygon.

Representing Polygon Meshes

In this section, we discuss three polygon-mesh representations: explicit, pointers to a vertex list, and pointers to an edge list. In the *explicit* representation, each polygon is represented by a list of vertex coordinates:

P = «Xl' Yl' *ZI), (_, Y2, Z2), . . ., (X., Y., Z.»*.

The vertices are stored in the order in which they would be encountered traveling around the polygon. There are edges between successive vertices in the list and. between the last and first vertices. For a single polygon, this is space-efficient; for a polygon mesh, however, much space is lost because the coordinates of shared vertices are duplicated, Even worse, there is no explicit representation of shared edges and vertices, For instance, to drag" vertex and all its incident edges interactively, we must find all polygons that share. vertex, This requires comparing the coordinate triples of one polygon with those of all ot polygons, The most efficient way to do this would be to sort all *N* coordinate triples, but this is at best an *Nlog.)Y* process, and even then there is the danger that the same vertex migQ,t_;_J due to computational roundoff, have slightly different coordinate values in each polygon, _Qjf,

a correct match might never be made. "_ ,'!Iii

With this representation, displaying,the mesh either as filled polygons or as polygo!l'_

outlines necessitates transforming each vertex and clipping each edge of each polygon.}IfJ"

edges are being drawn, each shared edge is drawn twice; this causes problems on pen_

plotters, film recorders, and vector displays due to the overwriting. A problem may also be' created on raster displays if the edges are drawn in opposite directions, in which case extra pixels may be intensified.

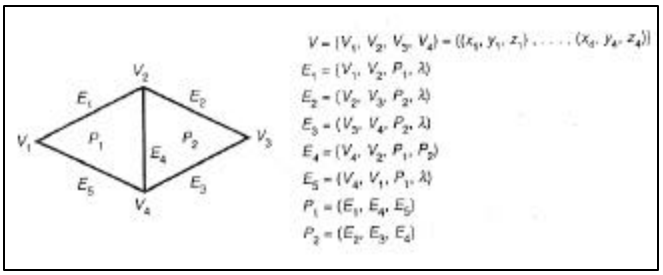
Polygons defined with *pointers to a vertex list,* the mf'thod used by SPHIGS, have each vertex in the polygon mesh stored just once, in the wrtex list $V = «XI'$ YI' ZI)' . . ., $(xn, Yn, z,,))$. A polygon is defined by a list of indices (or pointers) into the vertex list. A polygQn' made up of vertices 3, 5, 7, and 10 in the vertex list would thys be represented as $P = (3,5, 7, 10)$.

This representation, an example of which is shown in Fig. 11.3, has several advantages over the explicit polygon representation. Since each vertex is stored just once, considerable space is saved. Furthermore, the coorctinates of a vertex can be changed easily. On the other hand, it is still difficult to find polygons that share an edge, and shared polygon edges are still drawn twice when all polygon outlines are displayed. These two problems can be eliminated by representing edges explicitly, as in the next method.

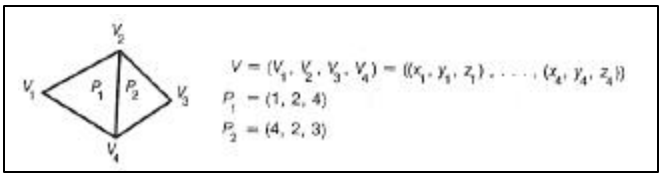
When defining polygons by *pointers to an edge list.* we again have the vertex list V, but represent a polygon as a list of pointers not to the vertex list, but rather to an edge list, in' which each edge occurs just once. In turn, each edge in the edge list points to the two vertices in the vertex list defining the edge, and also to the one or two polygons to which the' edge belongs. Hence, we describe a polygon as $P = (E \setminus, . . ., En)$, and an edge as $E = (VI, V2'$ PI, $P2)$. When an edge belongs to only one polygon, either PI or $P2$ is null. Figure 11.4 shows an example of this representation.

Polygon outlines are shown by displaying all edges, rather than by displaying all polygons; thus, redundant clipping, transformation, and scan conversion are avoided. Filled polygons are also displayed easily. In some situations, such as the description of a '3D honeycomblike sheet-metal structure, some edges are shared by three polygons. In such cases, the edge descriptions can be extended to include an arbitrary number of polygons: $E = (VI, Vz, PI, Pz, . . ., Pn)$.

In none of these three representations (Le., explicit polygons, pointers to vertices, pointers to an edge list), is it easy to determine which edges are incident to a vertex: All edges must be inspected. Of course, information can be added explicitly to permit determining such relationships. For instance, the winged-edge representation used by Baumgart [BAUM75] expands the edge description to include pointers to the two adjoining edges of each polygon, whereas the vertex description includes a pointer to an (arbitrary) edge incident on the vertex, and thus more polygon and vertex information is available.



polygon meshes defined with edge list for each polygon



polygon meshes defined with indexes into a virtual list

Consistency of Polygon-Mesh Representations

Polygon meshes are often generated interactively, such as by operators digitizing drawings, so errors are inevitable. Thus, it is appropriate to make sure that all polygons are closed, all edges are used at least once put not more than some (application-defined) maximum, and each vertex is referenced by at least two edges. In some applications, we would also expect the mesh to be completely connected (any vertex can be reached from any other vertex by moving along edges), to be topologically planar (the binary relation on vertices defined by edges can be represented by a planar graph), or to have no holes (there exists just one boundary-a connected sequence of edges each of which is used by one polygon).

Of the three representations discussed, the explicit-edge scheme is the easiest to check for consistency, because it contains the most information. For example, to make sure thltt all edges are part of at least one but no more than some maximum number of polygons, the code in Fig. can be used.

This procedure is by no means a complete consistency check. For example, an edge used twice in the same polygon goes undetected. A similar procedure can be used to make sure that each vertex is part of at least one polygon; we check whether at least two different edges of the same polygon refer to the vertex. Also, it should be an error for the two vertices of an edge to be the same, unless edges with zero length are allowed.

The relationship of "sharing an edge" between polygons is a binary equivalence relation and hence partitions a mesh into equivalence classes called *connected components.*

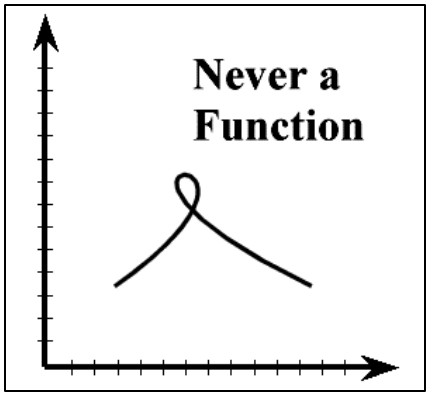
LESSON 31
PARAMETRIC CUBIC CURVES

Today Topic

- **Parametric Cubic Curves**

Parametric Curves

Parametric curves are very flexible They are not required to be functionsCurves can be multi-valued with respect to any coordinate system (a functions return one unique value for a given entry)



Parameter count generally gives the objects's dimension
Hyperplane : (n-1 parameter) in space of dimension n ,

Decouples dimension of object from the dimension of the space

r(u) (instead of [x(u), y(u), z(u)]) : vector-valued parametric curve

Notion of finite or infinite length : close (could always be bring back to [0 ; 1]) or open interval for u

In Cartesian space, a point is defined by distances from the origin along the three mutually orthogonal axes x, y, and z. In vector algebra, a point is often defined by a *position vector* **r**, which is the displacement with the initial point at the origin. The path of a moving point is then described by the position vectors at successive values of the parameter, say u Î Â. Hence, the position vector **r** is a function of u, i.e., **r** = **r**(u). In the literature, **r**(u) is called the *vector-valued parametric curve*. Representing a parametric curve in the vector-valued form allows a uniform treatment of two-, three-, or n-dimensional space, and provides a simple yet highly expressive notation for n-dimensional problems. Thus, its use allows researchers and programmers to use simple, concise, and elegant equations to formalize their algorithms before expressing them explicitly in the Cartesian space. For these reasons, a vector-valued parametric form is used intensively for describing geometric shapes in computer graphics and computer aided geometric design.

It should be noted that the curve **r**(u) is said to have an *infinite length* if the parameter is not restricted in any specific interval, i.e., u ∈ (-∞,+∞).

Conversely, the curve **r**(u) is said to have a *finite length* if u is within a closed interval, for example, u ∈ [a,b] where a, b ∈ ℜ.

Given a finite parametric interval [a,b], a simple reparametrization t = (u-a)/(b-a) would normalize the parametric interval to t ∈ [0,1].

A comprehensive study on curves, including polynomial parametric curves, is beyond the scope of this chapter. Interested readers may find such information in many text books on algebraic curves. In this chapter, we discuss only two types of parametric curves, namely *Bézier curves* and *B-spline curves*. In particular, we are concerned with the representation of Bézier and B-spline curves as well as some essential geometric processing methods required for displaying and manipulating curves.

Representation of Curves

Possible representation of curves: explicit, implicit and parametric

Explicit representation

- curve in 2D y=f(x)
- curve in 3D y=f(x), z=g(x)
- easy to compute a point (given parameters)
- multiple values of y and z for a single x is impossible (e.g., circle where $y = \pm\sqrt{r^2 - x^2}$ has 2 or 0 values)
- how to represent an infinite slope?
- Implicit representation
- curve in 2D: F(x,y)=0
- line: ax+by+c=0
- circle: 0 2 2 2 = - + *r y x*
- surface in 3D: F(x,y,z)=0
- plane: ax+by+cz+d=0

- F(x,y,z)=0 describes a 3D object $\begin{cases} F(x,y,z) < 0, \text{inside} \\ F(x,y,z) = 0, \text{on surface} \\ F(x,y,z) > 0, \text{inside} \end{cases}$

- However, computing a point is difficult (it is easy to determine where any point lies in terms of that curve).

Parametric Representation

- curves: single parameter (e.g] 1 , 0 [∈ *u*)
- x=X(u), y=Y(u), z=Z(u)
- tangent is simply derivatives of the above functions

Parametric cubic curves

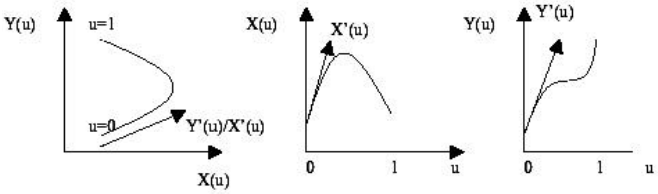
- Why cubic?
- Curves of lower order commonly have too little flexibility
- Curves of higher order are unnecessarily complex and can introduce certain artifacts
- Cubic curves offer a good trade-off between complexity and flexibility

$$\begin{cases} X(u) = a_3u^3 + a_2u^2 + a_1u + a_0 \\ Y(u) = b_3u^3 + b_2u^2 + b_1u + b_0 \\ Z(u) = c_3u^3 + c_2u^2 + c_1u + c_0 \end{cases}$$

- compact representation

$$X(u) = [u^3, u^2, u, 1]\begin{bmatrix} a_3 \\ a_2 \\ a_1 \\ a_0 \end{bmatrix} \quad \text{i.e., } \begin{matrix} X(u) = UA \\ Y(u) = UB \\ Z(u) = UC \end{matrix}$$

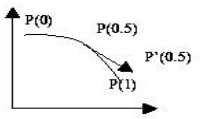
- derivatives are easy: $\frac{dX(u)}{du} = [3u^2 \quad 2u \quad 1 \quad 0]A$
- think of u as a time parameter



Question

Find the curve parameters given that the user specifies the following 4 parameters:

- 2 end points
- midpoint
- tangent at midpoint



$$\begin{bmatrix} X(0) \\ X(0.5) \\ X'(0.5) \\ X(1) \end{bmatrix} = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0.5^3 & 0.5^2 & 0.5 & 1 \\ 3(0.5)^2 & 2(0.5) & 1 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix} A$$

- $G_x = BA \Rightarrow A = B^{-1}G_x$
- since $X(u)=UA$, we have $X(u) = UB^{-1}G_x \Rightarrow X(u) = UMG_x$, where M is called the basis matrix, which in this case is

$$M = \begin{bmatrix} -4 & 0 & -4 & 4 \\ 8 & -4 & 6 & -4 \\ -5 & 4 & -2 & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

- we can then rewrite our equation

$X(u) = UMG_x = [f_1(u), \quad f_2(u), \quad f_3(u), \quad f_4(u)]G_x$,
where

$f_1(u) = -4u^3 + 8u^2 - 5u + 1$
$f_2(u) = -4u^2 + 4u$
$f_3(u) = -4u^3 + 6u^2 - 2u$
$f_4(u) = 4u^3 - 4u^2 + 1$

(similarly for Y and Z).

Parametric Continuity (for different curves p and q)

- C0: matching endpoints p(1)=q(0)
- C1: matching derivatives (with same magnitude) p'(1)=q'(0)
- C2: 1st and 2nd derivatives are equal
- Cn: nth derivatives are equal

Geometric Continuity

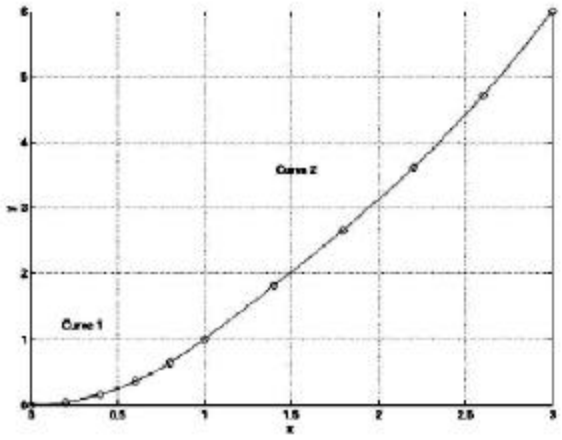
- G0=C0
- G1: matching derivatives (not necessarily magnitude)
- G2: 1st and 2nd derivatives proportional
- Gn: nth derivatives are proportional

Example: Determine if the following curves are C_0, C_1, G_1 continuous

$A(u) = (u, u^2)$
$B(u) = (2u+1, u^2+4u+1)$ for $u \in [0,1]$

A(1)=(1,1) and B(0)=(1,1), thus they're C_0 continuous

A'(1)=(1,2) and B'(0)=(2,4), magnitudes are different but derivatives are proportional, thus they're G_1 continuous, but not C_1.



Note the difference in speed between curves 1 and 2.

LESSON 32
BEZIER CURVES

Today Topic

- **Bezier Curves**

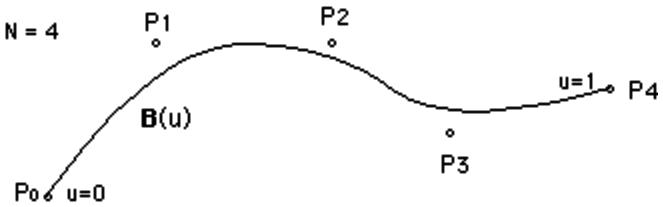
Bezier Curves

The following describes the mathematics for the so called Bézier curve. It is attributed and named after a French engineer, Pierre Bézier, who used them for the body design of the Renault car in the 1970's. They have since obtained dominance in the typesetting industry and in particular with the Adobe Postscript and font products.

Consider N+1 control points pk (k=0 to N) in 3 space. The Bézier parametric curve function is of the form

$$\mathbf{B}(u) = \sum_{k=0}^{N} pk \frac{N!}{k!\,(N-k)!} u^k (1-u)^{N-k} \quad \text{for } 0 \le u \le 1$$

B(u) is a continuous function in 3 space defining the curve with N discrete control points P_k. u=0 at the first control point (k=0) and u=1 at the last control point (k=N).



Notes

- The curve in general does not pass through any of the control points except the first and last. From the formula **B**(0) = P_0 and **B**(1) = P_N.
- The curve is always contained within the convex hull of the control points, it never oscillates wildly away from the control points.
- If there is only one control point P_0, ie: N=0 then **B**(u) = P_0 for all u.
- If there are only two control points P_0 and P_1, ie: N=1 then the formula reduces to a line segment between the two control points.

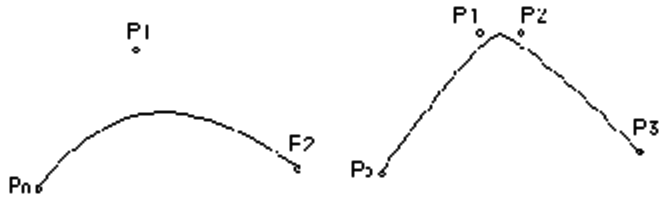
$$\mathbf{B}(u) = \sum_{k=0}^{1} pk \frac{1}{k!\,(1-k)!} u^k (1-u)^{1-k} = p0 + u(p1 - p0)$$

- the term

$$\frac{N!}{k!\,(N-k)!} u^k (1-u)^{N-k}$$

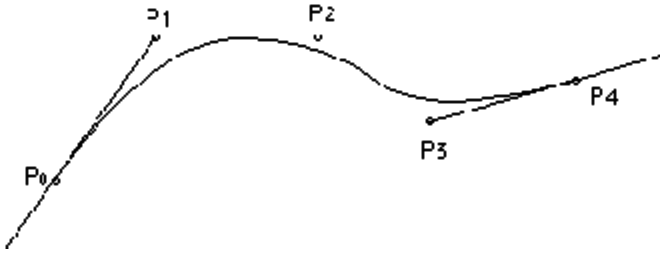
is called a blending function since it blends the control points to form the Bézier curve.

- The blending function is always a polynomial one degree less than the number of control points. Thus 3 control points results in a parabola, 4 control points a cubic curve etc.
- Closed curves can be generated by making the last control point the same as the first control point. First order continuity can be achieved by ensuring the tangent between the first two points and the last two points are the same.
- Adding multiple control points at a single position in space will add more weight to that point "pulling" the Bézier curve towards it.



- As the number of control points increases it is necessary to have higher order polynomials and possibly higher factorials. It is common therefore to piece together small sections of Bézier curves to form a longer curve. This also helps control local conditions, normally changing the position of one control point will affect the whole curve. Of course since the curve starts and ends at the first and last control point it is easy to physically match the sections. It is also possible to match the first derivative since the tangent at the ends is along the line between the two points at the end.

  Second order continuity is generally not possible.



- Except for the redundant cases of 2 control points (straight line), it is generally not possible to derive a Bézier curve that is parallel to another Bézier curve.
- A circle cannot be exactly represented with a Bézier curve.
- It isn't possible to create a Bézier curve that is parallel to another, except in the trivial cases of coincident parallel curves or straight line Bézier curves.
- Special case, 3 control points

$\mathbf{B}(u) = P_0 * (1 - u)^2 + P_1 * 2 * u (1 - u) + P_2 u^2$

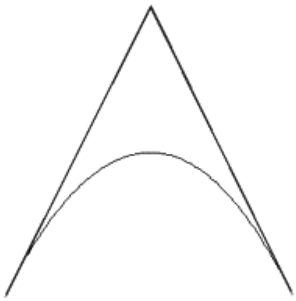
- Special case, 4 control points

$\mathbf{B}(u) = P_0 * (1 - u)^3 + P_1 * 3 * u * (1 - u)^2 + P_2 * 3 * u^2 * (1 - u) + P_3 * u^3$

Bézier curves have wide applications because they are easy to compute and very stable. There are similar formulations which are also called Bézier curves which behave differently, in particular it is possible to create a similar curve except that it passes through the control points. See also Spline curves.

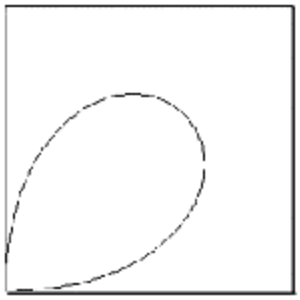
Examples

The pink lines show the control point polygon, the grey lines the Bézier curve.

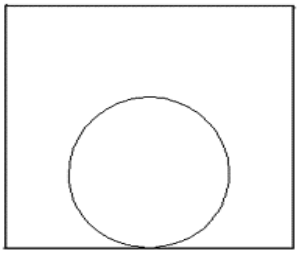


The degree of the curve is one less than the number of control points, so it is a quadratic for 3 control points. It will always be symmetric for a symmetric control point arrangement.

The curve always passes through the end points and is tangent to the line between the last two and first two control points. This permits ready piecing of multiple Bézier curves together with first order continuity.



The curve always lies within the convex hull of the control points. Thus the curve is always "well behaved" and does not oscillating erratically.



Closed curves are generated by specifying the first point the same as the last point. If the tangents at the first and last points match then the curve will be closed with first order continuity.. In addition, the curve may be pulled towards a control point by specifying it multiple times.

A cubic Bezier curve is simply described by four ordered control points, p0, p1, p2, and p3. It is easy enough to say that the curve should "bend towards" the points. It has three general properties:

1. The curve interpolates the endpoints: we want the curve to start at p0 and end at p3.
2. The control points have local control: we'd like the curve near a control point to move when we move the control point, but have the rest of the curve not move as much.
3. The curve stays within the convex hull of the control points. It can be culled against quickly for visibility culling or hit testing.

A set of functions, called the Bernstein basis functions, satisfy the three general properties of cubic Bezier curves.

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad \text{for } 0 \le i \le n$$

If we were considering general Bezier curves, we'd have to calculate *n choose i*. Since we are only considering cubic curves, though, $n = 3$, and i is in the range [0,3]. Then, we further note the *n choose i* is the *i*th element of the *n*th row of Pascal's traingle, {1,3,3,1}. This value is hardcoded rather than computed in the demo program.

Bezier Patches

Since a Bezier curve was a function of one variable, *f(u)*, it's logical that a surface would be a function of two variables, *f(u,v)*. Following this logic, since a Bezier curve had a one-dimentional array of control points, it makes sense that a patch would have a two-dimensional array of control points. The phrase "bicubic" means that the surface is a cubic function in two variables - it is cubic along *u* and also along *v*. Since a cubic Bezier curve has a 1x4 array of control points, bicubic Beizer patch has a 4x4 array of control points.

To extend the original Bernstein basis function into two dimension, we evaluate the influence of all 16 control points:

$$\sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij} B_i^3(u) B_j^3(v)$$

The extension from Bezier curves to patches still satisfies the three properties:

1. The patch interpolates p00, p03, p30, and p33 as endpoints.
2. Control points have local control: moving a point over the center of the patch will most strongly affect the surface near that point.
3. The patch remains within the convex hull of its control points.

Implementing Bezier Patches

Rendering a Bezier patch is more complicated than rendering a Bezier curve, even when doing it in the simplest possible way. With a Bezier curve, we could just evaluate strips of points and render a line of strips. With a patch, we need to evaluate strips of points and render triangle strips. Also, with a patch we have to worry about lighting. Even with the simplest lighting method, we need to light each vertex, which means we need each of the vertex's normal. So for every *(u,v)* pair, we need to solve the point on the surface, and then solve for its normal.

To find the normal of a point on the surface, we can take the derivative of the surface with respect to either *u* or *v*, which yields the tangent vectors to the surface in the direction of either *u* or *v*. If we find both of these tangents, we know that they both lie in the plane tangent to the surface. Their cross product yields the surface normal after we normalize it. This renders the Bezier patches without optimization.

To find $df(u,v)/du$.

$$\frac{df(u)}{du} =$$

$$\frac{d(\sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij}B_i^3(u)B_j^3(v))}{du} =$$

$$\sum_{i=0}^{3}\sum_{j=0}^{3} p_{ij}\frac{dB_i^3(u)}{du}B_j^3(v)$$

The same holds for $df(u,v)/dv$.

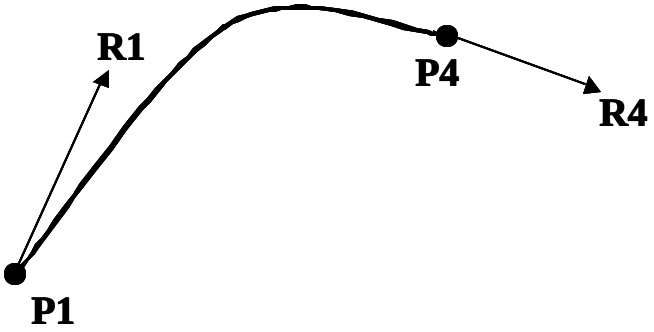
Notes:

LESSON 33
HERMITE CURVES

Today topic

- **Hermite Curves**

Hermite Curves



The Hermite curve family is a parametric cubic curve described by 2 endpoints and a tangent vector at each of those endpoints. The endpoints are described as P1 and P4 (this will be understood when we get to curves with intermediary control points). The tangent vectors are usually described as R1 and R4.

Figure 1 - A Hermite Curve is defined by two endpoints and two end-vectors. The vectors influence the curve as the curve is drawn from one endpoint to the other.

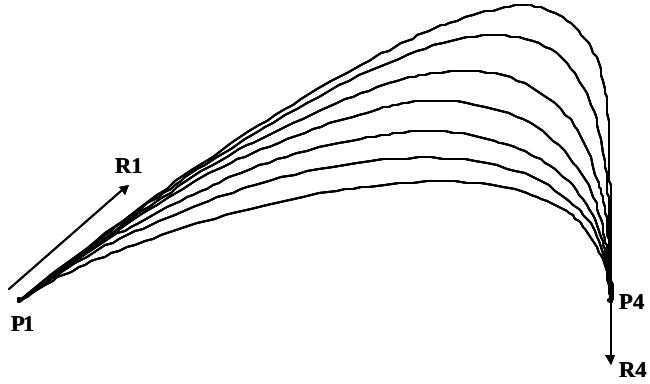


Figure 2 - Each of the curves below are drawn from the same endpoints. The only difference is in the magnitude of vector R1. The bigger curves have a larger R1 vector while the smaller curves on the bottom have a smaller R1 vector.

Math behind calculating Hermite Curves

The equation for Hermite curves is shown below. It is simply an extension of the general parametic curve equation stated above, but the Hermite family has its own basis matrix and unique geometry vectors made up of the endpoints and tangent vectors.

$$Q(t) = T \bullet M_H \bullet G_H = \begin{bmatrix} t^3 & t^2 & t & 1 \end{bmatrix} M_H \begin{bmatrix} P_1 \\ P_4 \\ R_1 \\ R_4 \end{bmatrix}$$

Basis Matrix: The basis matrix is shown above as M_H. To see how the Basis matrix is derived see Foley, Van Dam, Feiner, and Hughes p. 484.

The Hermite basis matrix is as follows:

$$M_H = \begin{bmatrix} 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 1 \\ 0 & 0 & 1 & 0 \\ 3 & 2 & 1 & 0 \end{bmatrix}^{-1} = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}$$

Geometry Vector: The geometry vector is unique to every set of Endpoints and tangent vectors. The geometry vector is shown above and contains the two endpoints followed by the two tangent vectors.

Notes:

LESSON 34
INTRODUCTION TO ANIMATION AND DESIGN TRICKS

Today Topic

- **Intoroduction to Animation**
- **Design tricks**

Animation

To animate is, literally, to bring to life. Although people often think of animation as synonymous with motion, it covers all changes thar have a visual effect. It thus includes the time-varying position (motion dynamics), shape, color, transparency, structure, and texture of an object (update dynamics), and changes in lighting, camera position, orientation, and focus, and even changes ‘of rendering technique.

Animation is used widely in the” entertainment industry, and is also being applied in education, in industrial applications such as control systems and heads-up displays and flight simulators for aircraft, and in scientific research. The scientific applications of computer graphics, and especially of animation, have come to be grouped under the heading scientijir visualization. Visualization is more than the mere application of graphics to science and engineering, however; it can involve other disciplines, such as signal processing, computational geometry, and database theory. Often, the animations in scientific visualization are generated from simulations of scientific phenomena. The results of the similations m-ay be large datasets representing 20 or 3D data (e.g., in the case of fluid-flow simulations); these data are converted into images that then constitute the animation. At the other extreme, the simulation may generate positions and locations of physical objects, which must then be rendered in some form to generate the animation. This happens, for example, in chemical simulations, where the positions and orientations of the various atoms in a reaction may be generated by simulation, but the animation may show a ball-and-stick view of each molecule, or may show overlapping smoothly shaded spheres representing each atom. In some cases, the simulation program will contain an embedded animation language, so that the simulation and animation processes are simultaneous.

If some aspect of an animation changes too quickly relative to the number of animated frames displayed per second, temporal aliasing occurs. Examples of this are wagon wheels that apparently turn backward and the jerky motion of objects that move through a hirge field of view in a short time. Videotape is shown at 30 frames per second (fps) , and:j_ photographic film speed is typically 24 fps, and both of these provide adequate results many applications. Of course, to take advantage of these rates, we must create a new image;(f for each videotape or film frame. If, instead, the animator records each image on two,/ videotape frames, the result will be an effective 15 fps, and the motion will appear jerkier:1

Some of the animation techniques described here have been ‘partially or completely’ implemented in hardware. Architectures supporting basic animation in real time are essential for building flight simulators and other real-time control systems; some of these architectures were discussed in Chapter 18.

Traditional animation (i.e., noncomputer animation) is a discipline in itself, and we do not discuss all its aspects. Here, we concentrate on the basic concepts of computer-based1

animation, and also describe some state-of-the-art systems. We begin by discussing conventional animation and the ways in which computers have been used to assist in its creation. We then move on to animation produced principally by computer. Since much of this is 3D animation, many of the techniques from traditional 2D character animation no longer apply directly. Also, controlling the course of an animation is more difficult when the animator is not drawing the animation directly: it is often more difficult to describe how to do something than it is to do that action directly. Thus, after describing various animation languages, we examine several animation control techniques. We conclude by discussing a few general rules for animation, and problems peculiar to animation.

All animation is, in a sense, a trick. We show the viewers a series of still pictures of inanimate objects and expect them to see a moving scene of something that is alive. Most viewers are aware of this trick. The really good tricks, however, are those which wouldn’t be noticed unless they are pointed out. In these notes I am going to point out various tricks that I’ve used to good effect in production of several “realistic” space simulation movies at JPL, and in the production of several, more schematic, animations for The Mechanical Universe and Project Mathematics! Some of these tricks are small, some are big. Some might actually be useful to other people.

The concept of “trick” in some sense implies chaos. Tricks don’t always admit to categorization. Nevertheless, due perhaps to my academic background, I will try to divide them into somewhat related topics.

Design Tricks

The basic intent of an animation is to communicate something. Sometimes the most obvious animation is not sufficient to get the desired idea across. Tricks in this category relate to how the design of an animation can improve communication.

Attraction Tricks

One of the important aspects of design, both for still and for moving images, is the direction of the viewer’s attention to what the designer considers the important parts of the image. This is especially important for moving images since they are on the screen for a short time; the viewer typically does not have the option of studying them for a long time. Furthermore, my educational animations typically consist of some action, a pause for the viewer to absorb the intent, and then some new action. It is important to get the viewer looking at the place where the new action will occur before it is all over. I’ll list some simple,

and fairly obvious, ways to do this roughly in inverse order of subtlety.

- Appearing and Disappearing

The eye is attracted to change on the screen. What could be more dramatic than something changing from existence to non-existence (or vice versa). I use this a lot in my animation of mathematical proofs. A traditional textbook proof contains a diagram cluttered with a lot of labels and construction lines. The reader continually flips back and forth between the diagram and the text trying to relate them; "line AB is parallel to CD and intersects at point P at angle alpha". In an animated proof I only need to show auxiliary lines when they are needed. Often I don't need to show many labels at all and can therefore keep the diagram as simple as possible. The narration then can go something like "this line [line appears] is parallel to this one [other line appears] and intersects at this point [point appears] at this angle [angle arc appears, lines and point disappear]".

Incidentally, having objects appear and disappear from a screen in one frame time produces an overly percussive effect. I usually use a 5 frame fade-in for all appearing and disappearing objects.

- Blinking

Simply blinking an object is rather unsubtle and I try to use this technique as little as possible but you cannot deny that having something blink before it moves gets your attention. I've found that a blink rate of about 3 cycles per second is best. This can be the obvious 5 frames on and 5 frames off, but it's sometimes more interesting to try 7 frames on and 3 off. Actually the difference is perhaps too subtle to notice.

- Anticipation and Overshoot

Conventional animation makes great use of this; Wiley Coyote will rear back and pause for a second before dashing off screen. This can be seen as an animator's version of the mathematical Gibb's phenomenon; when a function is approximated with low frequency sine waves, the function overshoots at the beginning and end of an abrupt transition. In a similar manner I try to anticipate actions in dancing equations by having objects back up a bit before moving in a desired direction. The minimum values for this to be effective seems to be a 7 frame anticipation followed by a 5 frame pause, followed by the final action. I also find that, for my work the overshoot portion seems undesirable and distracting.

- The See-saw Effect

Many mathematical demonstrations consist of a series of transformations that keep some quantity constant. For example a shape keeps the same area if it is sheared, or translated, or rotated. I sometimes lead the viewer into some such transformation by "rubbing the transformation back and forth" a bit. This is somewhat like anticipation with several cycles of oscillation of transformation before the object begins to move.

- Parallel Action

I often need to point up the connection between two things on different parts of the screen. For example, in a substitution an algebraic term appears in two different equation. I attract attention to them by having them both shake up and down at the same time.

In other situations I need to show how an Algebraic quantity ties in with its Geometric meaning. Here I typically have a label on a diagram "ghost out" of the diagram and fly into the equation.

In contrast, there are sometimes situations where I want to point out the differences between objects. For example, I typically have two types of things that are might move: geometric parameters of a diagram (like angles or lengths of sides) and annotations (like labels and equations). I want to try to distinguish between them via motion as well as appearance. I do this by having motions of parameters interpolate linearly (at a constant speed) and motions of annotations do smooth ease-in and ease-out motions. This gives a more mechanical motion to the mechanical parts of the diagram.

- Tension and Release

The mood of any scene goes through highs and lows. There are several ways to effect this with both sound and visuals. Musicians can set up tensions with dissonances or dominant 7th chords and can bring about a sense of repose by resolving the chord back to the tonic. Likewise, visually you can create tension by making shapes seem unbalanced, nearly tipping over. Release comes from objects firmly placed.

I have used this technique to show intermediate results in equations as unbalanced, with the equation tipped slightly, and then literally balancing the equation when the equation is finally solved.

- Hesitation

The final step of a proof is a sort of punch line to a scene. To get the viewer involved I don't just move the shapes directly in place; I sometimes make them pause a bit before final positioning to build up anticipation.

Distraction Tricks

Stage magic is a process of distracting the audience from the secret the magician is trying to disguise. In the same manner it is sometimes necessary to distract the viewer from something on the screen. This might be a glitch in the animation rendering, or it might be a cheat that the animator is using to avoid some lengthy or complicated computations. Here are some examples

- The Old Switcheroo

Often different models are needed for the same object at different points of an animation. I have used these tricks in the following situations

Voyager -The complete database of the Voyager spacecraft was too big for early versions of my renderer. I created two versions, one containing only the polygons visible on the front side and one containing only those visible on the back side. I switched models from the front-side version to the back-side version when the spacecraft was panned off the screen.

Planets - Moons and planets that are so small that they project into less than 2 pixels are replaced by a simple anti-aliased dot of the same color as the average of the texture map. I made sure that the transition was minimized by adjusting the size of the dot to approximately match the size of the sphere at the transition.

Bomb - I drew an exploding spherical bomb (used to illustrate Center of Mass) with my special case planet drawing program until it exploded. There was a flash during the explosion frame when I replaced the bomb with an irregular polygonal model for the pieces to fly apart.

Surface of Mimas - A similar database/rendering switch was used in a scene depicting a flight over Saturn's moon Mimas. When viewed near the North pole I rendered the moon as a bump mapped sphere. When viewed near the large crater at the equator I rendered it as a brute force polygon mesh. The transition was made manually by first generating the frames near the pole, stopping frame production, and editing the control file to cause it to invoke a different rendering program, and then re-starting production. The transition was done when flying over the night side of the moon where the image was too dark to notice the slight change in appearance.

- Covering Mistakes

An error in the database of the Mimas crater made another trick necessary. Some of the polygons in the rim of the crater were accidentally deleted during the production process. We didn't have time to chase down the problem and re-render all the bad frames. I was however able to re-render some of the frames near the problem. I picked a time in the animation when the sun just pops over the limb of the moon to make the switch. The eye is so surprised by the sudden change in illumination that the viewers don't notice that a notch appears in the crater rim.

Timing Tricks

These tricks pertain to how long you make actions take to occur.

- Speed Adjustment

A lot of physical actions happen too quickly to see. A simple solution is to slow down the time scale. Just as you scale size of an object to fit on screen, you can scale time to fit to a sound track. I especially needed to do this for several scenes of falling objects: apples, amusement park rides, etc. These hit the ground much more quickly than would allow time for the desired narration.

- Logarithmic Zooms

When flying in to objects that vary greatly in scale it's useful to animate the logarithm of the distance rather than the distance directly. This was built in to the space flyby simulation program and I used it explicitly in several other space simulations.

- When to Double/Single Frame

A common cheat in animation is to do double framing, that is, to only render every other frame and to record each rendered frame twice. We all expect that single framing is preferable to double framing, its only disadvantage is that single framing takes longer to render. There is another effect of double framing however. A motion that is double framed seems to move faster than one that is single framed, even if they take an identical amount of wall-clock time to take place. Double framing is therefore sometimes used by conventional animators to add liveliness to scene.

- Rhythm

My current animations system allows me to specify keyframes either dynamically or numerically via a spreadsheet like display. Being numerically inclined I usually use the latter and have developed the habit of making keyframe numbers be multiplies of 5, just to make the numerical display look prettier. I started doing this for rough tests, expecting to later change the numbers to more general values determined from observing the timing of the rough tests. The composer for the Mechanical Universe project, however, told me she liked the timing of these animations because they have a rhythm to them. The motion always started and stopped to a beat of 360 beats per minute. Now that we are using a generic music library where a lot of the music is composed to 120 beats per minute, the 5 frame timing constraint makes the animation fit to the music quite well without doing any explicit matching.

- Overlapping Action

In The Illusion of Life, Frank Thomas and Ollie Johnston pointed up the technique of making an animation more alive by having various actions overlap in time. Whenever I've tried this for my algebraic ballets I haven't been satisfied with it. It seems that for my applications the motion is much clearer if all the actions for a particular transformation end at the same time. So even if I make the motion of several objects start at different times I try to make them end at the same time.

Notes:

LESSON 35
DESIGN TRICKS (CONTD…)

Today Topic

· **Design Tricks Contd.**

Motion Enhancement

Some sorts of motion are hard to convey without exaggerating them in some way. Here are some examples.

- Falling Bodies

Suppose you want to show something falling. It leaves the screen almost immediately. So you decide to track it as it falls. Now it appears stationary on the screen. How do you give the impression of something falling continuously? Put some texture in the background that scrolls up as you track the object. The texture doesn't even have to be particularly realistic. I did this in two MU scenes, one showing a falling anvil and one showing a falling oil drop (for the Millikan oil drop experiment).

You could also give impression of a falling object being tracked by a human cameraman by adding some random fluctuation to the position on the screen. This is an example of how our minds have been trained to react to the visual artifacts of a particular technology; we tend to date images as being made when that technology was common. Other examples are the use of black and white images to give the impression of pre-50's movies, or of using grainy jumpy camera work to give the impression of home movies of the 50's and 60's.

- Rolling Ball

Here's the problem: show a ball rolling down an inclined plane. The design of the scene was supposed to mimic a drawing in Galileo's notebooks. This means the ball was to be drawn as just a circle, which wouldn't look much like it was rotating. Now, it turns out that I had a ball digitized from another scene that had a simple line on it to represent a highlight. When I rolled it down the plane, the highlight rotated with the ball, looking just like a mark on the surface of the ball, and gave a nice impression of something rolling. If I had planned this from the first I wouldn't have tried this because I would have thought a mark near the edge of the ball would just look weird and make it look lopsided.

· The Spinning Top

A scene from the program on angular momentum required a 3D view of a spinning top. Again we have the problem of a symmetric object spinning. The solution I explicitly used was to place a pair of black marks near the top of the top in a sort of plus-sign shape. These provided the asymmetry needed to follow the rotation. There was another unintended trick that also helped though; the top was made of Gouraud shaded polygons, with Gouraud shaded highlights. Even though the number of polygons was large enough for a still frame to look quite smooth, it was small enough so that irregularities in the image, and particularly in the highlight, gave a nice impression of motion.

Implementation Tricks

Many animation systems, my own included, are based on some sort of keyframing system applied to a nested transformation scheme. The animator must design the transformation structure and then specify values for the transformations (i.e. translation, scale and rotations values) and the keyframe numbers for them to have those values. Here are some tricks on how to generate such animation control files.

Top Down Design

There are two ways to proceed; animate each keyframe completely and then proceed to the next keyframe, or animate the root level of the transformation tree for all keyframes and then animate the next highest level for all keyframes etc. Experience of myself and of several other people is that the latter of the two is easiest. That is, you animate the grosser motions of the centers of your objects first, using simple linear interpolation between the keyframes.

Blocking

In a related vein there is the subject of timing. The conventional technique is to have the narrator record the script first, time it and then generate the animation to the timed sound track. This doesn't work too well in our situation since we are modifying the wording of the narration right up to the last minute. I find it best to first lay out the entire sequence of events with no thought given to how much time it takes for each event to occur. This is like blocking out a play. This step often takes many iterations since I am still trying to figure out in which order to show things.

Only after the sequence is set do I go back and spread apart the keyframe numbers to specify timing. I generally replay an animation many times while reading the narration, repeatedly lengthening the time durations of the motions and pauses to make sure there's time for all the words. This doesn't always work since our narrator reads more slowly than me.

Fine Tuning

Only after top level motions and time durations are set do I add the detailed motion of sub-objects, e.g. limbs of a character. Finally I add anticipation and overshoot to the linear interpolation used for the first approximation.

Changing Connectivity

Another common problem concerns the need to change the structure of the transformation tree dynamically during an animation. A familiar example would be if John gives an apple to Mary. When John holds the apple it is best for it to be at the end of the transformation tree of John's arm. When Mary holds the apple you want it to be at the end of the arm in Mary's transformation tree. I have many similar situations during algebraic ballets when an expression is factored and the terms must change their association during the animation.

The common solution to this problem is to have the object appear in the transformation tree at both positions, and utilize some trick to make only one copy visible at any time. Most animators I know use the trick of animating the translation parameters of the undesired object to move it off screen. My particular solution is to use the opacity channel of my system; I simply make the undesired object transparent. This has the double advantages of optimizing the tree (transparent compound objects are skipped during my tree traversal) and avoiding having a translated object show up accidentally if the viewing direction changes.

In any event, I generally need to manually align the two versions of the object at the transition frame via their respective transformation trees so that they are at the same place during the transition.

Squash and Stretch

Squash and stretch are commonly used to give life to inanimate objects. The problems is that such objects are usually modeled centered on their center of mass. Any squashing requires a scale factor about this center. Animating the center and scale factor makes it difficult to, e.g. keep the bottom of the object on a table before it jumps off.

A nicer way is to provide pure positional handles on each side of the object. It's more intuitive to animate these two locations separately, typically with similar motions but just displaced in time and position. You can then turn this into a centered translation and scale factor by something like:

Xcenter = (Xtop-Xbottom)/2

Xscale = (Xtop+Xbottom)/2

Economization Tricks

Here are some ways to produce scenes cheaply where a full simulation of the situation would be too hard or too slow.

Soft Objects

Objects that are translucent or cloudy are hard to animate. For some projects I was able to get away with the following.

- Scaling and Fading

You can make a somewhat schematic explosion by using a 2D texture map of a hand drawn puff of smoke. Simply animate it getting larger while making it progressively more transparent.

- 3D Sparkles

A nice sparkling effect can be made by using a 3D model of lines radiating randomly from a center. You then make the transparency of each line interpolate from opaque at the center to transparent at the endpoints. Then give the resultant shape a large rotation velocity around a couple of axes. The result will be a spray of lines from the origin that is radically different from one frame to the next.

Notes:

LESSON 36
DESIGN TRICKS (CONTD…)

Today Topic

· **Design tricks Contd.**

Temporal Anti-Aliasing

To properly portray motion with film or video one needs to do motion blur. This is often a real pain as the rendering system needs to know, not just about the position of each object in the scene, but the speed, and perhaps even the acceleration. Here are some ways of doing this explicitly.

- Speed Lines and Streaks

Animators and even still cartoonists can enhance motion by explicitly drawing lines trailing a moving object. This is an intuitive form of temporal antialiasing that can be used to great effect when done explicitly with computers. I've done this with various moving particles, for example:

Rapidly moving particles in Saturn's rings -I calculated the location of the particle on the previous frame and on the subsequent frame and drew a line between them. A correct temporal anti-aliasing of such a line would have it opaque at the center and fade out to transparency at the two endpoints. For largely intuitive reasons I instead drew the line more comet shaped, opaque at the terminating end transparent at the initial end. This seemed to work nicely even though the bright end of the line was actually at a location one frame-time in the future.

A similar animation showed a proton moving in the magnetic field of Jupiter with a bright dot trailing back to a more transparent line showing its entire path in the past.

Atomic motion -Another scene involved showing atoms moving in an ideal gas. The use of streaks here has an additional purpose. I have found that the eyes' ability to judge speed differences seems to be a lot less acute than its ability to judge, e.g. size differences. For example if you saw two objects moving across the screen one at a time it would be harder to tell which one moved faster than if you saw two lines appearing on the screen and wanted to know which one was larger. Therefore if you want to compare speeds it's helpful to add streaks to translate speed perception into size perception. I enhanced this effect by making the streaks for the atoms be about 6 times larger than the movement due to one frame time. Even in the still images you get a nice sense of motion from this scene.

- The Spinning Earth

One scene from The Mechanical Universe portrays long term precession of the spin axis of the Earth. If one were to watch this in speeded up motion the appearance of the Earth would, of course, turn into a blur. I explicitly performed this temporal anti-aliasing by pre-processing the Earth texture map to horizontally blur it. Simply showing a globe with uniform equatorial streaks might just look like a stationary globe with stripes. I got around this by speeding up the rotation in steps. First the Earth spun slowly with no blurring, then rather fast with moderate blurring, then real fast with extreme blurring. Finally it was rendered with a completely horizontally blurred pattern. In this case, of course, the physical model of the globe didn't need to spin at all since it would have looked the same at any angle. The build up, however, from the earlier speeds gave a tremendous impression of speed to the final version.

Simple Simulations

Often, even if a complete physically based model of some scene is not feasible, making some portions of the motions physically based is possible. The animator can have some "handles" on parts of an object that are animated explicitly, these then serve as inputs to some simple simulation of, say, internal structure of an object. Also, you can derive explicit solutions to some of these equations and give the animator control of the endpoints of the motion. The computer can then calculate what accelerations are necessary under the force laws to arrive at these endpoints.

Easy simulations include the following force laws

- Gravity

F=constant

- Damped Oscillation

$$\mathbf{F} = - k1\,\mathbf{x} - k2\,\mathbf{v}$$

- Lennard-Jones Atomic force

$$\mathbf{F} = (k1 / r\text{^}8 + k2 / r\text{^}14)\,\mathbf{r}$$

This latter requires numerical integration of many particles, but can generate very interesting motions, as well as illustrate many principles of thermodynamics.

Production Tricks

The term "production" refers to the process of rendering the frames and recording them to tape or film. These tricks go beyond just design and implementation of a design. They are methods to get the animation physically made. Many of these tricks are not glamorous, they are mainly bookkeeping techniques to keep production from getting out of hand.

- Start at Frame 1000

I always start animations at frame 1000 instead of frame 0. This enables me to go back after the fact and prepend an animation with other frames if necessary without needing negative frame numbers. Since files for rendered frames typically have a frame number as part of the file name, negative numbers might make invalid file names. Four digit frame numbers also allows me to get a sorted listing of frame files and have them come out in order; it avoids the problem of, say, frame 10 coming before frame 2 in the filename sort sequence.

- Skip Identical Frames

My educational animations generally consist of motion sequences interspersed with pauses. In production it is, of course, silly to re-render frames that are identical during a pause. I have generated an automatic mechanism that scans the animation control file and detects frames for which all the rendering parameters are identical. It then outputs a command file to render the scene in sections that actually move. The recording program then also must interpret this frame sequence file to record individual frames during a move and to repeat a frame during pauses.

- Binary Search Rendering Order

In order to debug the rendering on a scene it is often useful to render several frames scattered through it. Then, if these are ok you can render the rest. It's a shame to re-do the frames you have already. One solution is to render every, say, 32 frames. Then render every 32 frames halfway between these, then every 16 frames halfway between these, then every 8 frames between these, etc. An advantage of this during time critical operations is that you can get a quadruple framed version done; then if you have time you can render the between frames to get a double framed version, and if you have still more time you can get a single framed version. This happened on the Cosmos DNA sequence. We originally only had time to to a quadruple framed version. When the producers saw it they gave us more time to do a double framed version. This leads to an old adage familiar to many production personnel "There's never enough time to do it right; there's always enough time to do it over".

- Multiple Machine Parallelism

This was a trick when I first did it in 1980 but it is basically the standard way to render, especially when machines are connected on a network.

Conclusion

Tricks are helpful and, as we all know, a "technique" is simply a "trick" that you use more than once.

Notes:

LESSON 37
KEY-FRAME ANIMATIONS

Today Topic

· **Key-frame Animations**

Key-frame Animations Introduction

In this part we will discuss animated objects. This library is a regular object library, however these objects include key frame animation. This animation is typically created in an outside 3D animation program and can be further edited in iSpace.

The following items will be covered

- Key-frame animation controls
- Key-frame animations created in trueSpace
- Key-frame animations created in iSpace
- Mini-Tutorial on inserting a key-frame animation into your scene.

Animation Controls

For each animation object there are two main controls

- The animation controls
- The animation bounding box

To preview the animation, select the parent cell (to which is animation is attached) and click to the play button in the tool bar. If you are viewing in top view, the animation will only play in the selected cell and may be clipped where it expends beyond the cell.

Left click animated objects to select them, as with any other iSpace object. An animated object can be scaled by dragging any edge of the bounding box (the animation gets scaled as well). Proportional scaling can be done by dragging any corner of the bounding box with both left and right mouse buttons. The animated object can be moved by left clicking inside the animation bounding box, holding, and dragging the box to the correct position. For more information, see the Objects chapter.

Creating and importing keyframe animations from trueSpace

The first category of animations can be created only in trueSpace and can be imported as an animated object through the File/ Load object menu. While trueSpace interpolates in-between frames from neighboring key-frames, iSpace makes frames **only** at key-frames without interpolation. This allows the user to control the size of the final GIF. The user has to create as many key-frames in trueSpace as he or she requires frames for the animated object in iSpace.

To import an animation from trueSpace:

- Create an animation
- Select the animated object
- Use Glue as Sibling to join all of the animated objects
- Save them as one .cob file
- Import to iSpace

Animated Objects in iSpace

iSpace contains libraries with pre-built animated objects. To use an animated object, load the key frames library, select an animated object. Left click, hold, and drag the object to the table. Release the mouse and the object will be inserted into the table panel. To preview the animation, make sure the animation is selected, then press the play button in the tool bar.

More on Key Frames

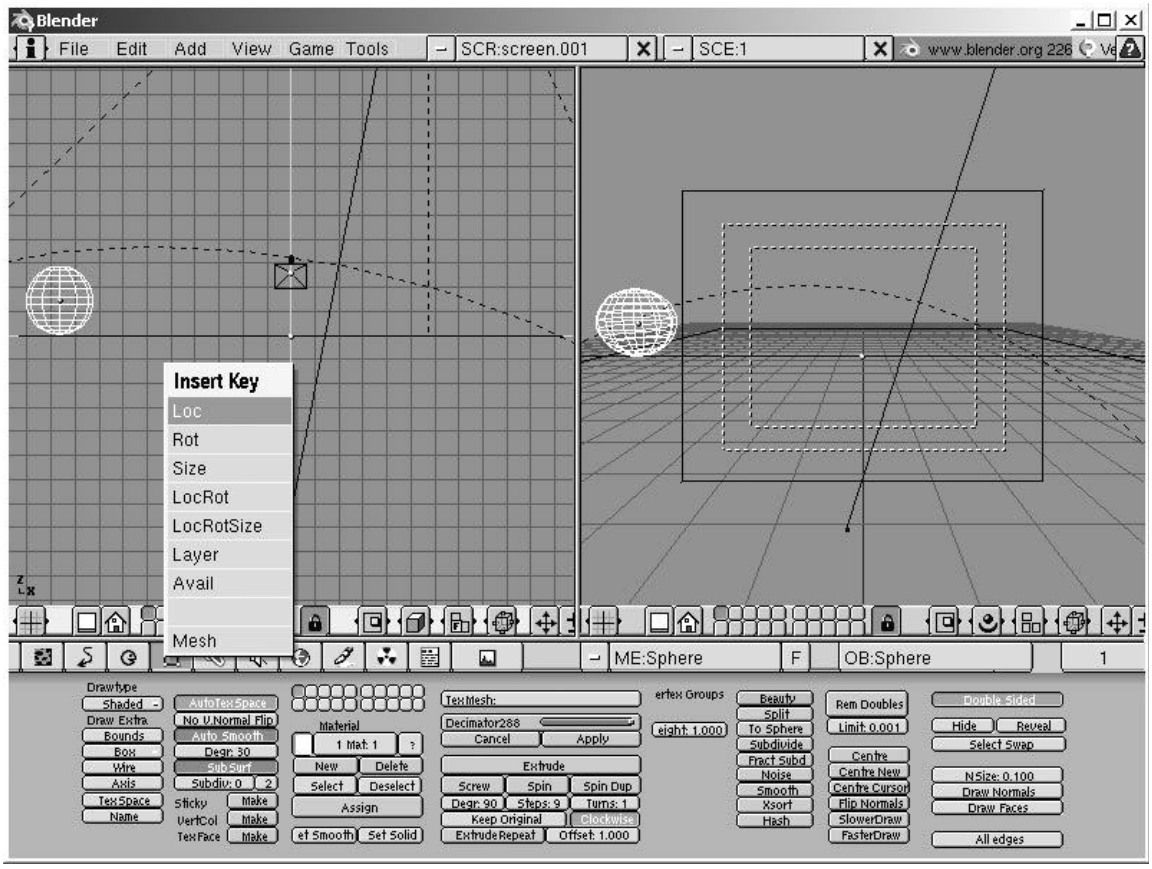
Do You know what is frames?

Frames

In the bar of the bottom panel you can see to the right there is a slightly longer box with a number in it. This is your frame counter. It tells you what frame you are currently on. You can change the frame you are on by either using the mouse to manipulate it like a normal Blender button... or you can also use the arrow keys. Pressing **Left** or **Right** will change the frame number by a count of 1 while pressing **Up** or **Down** will jump frames by a count of 10.

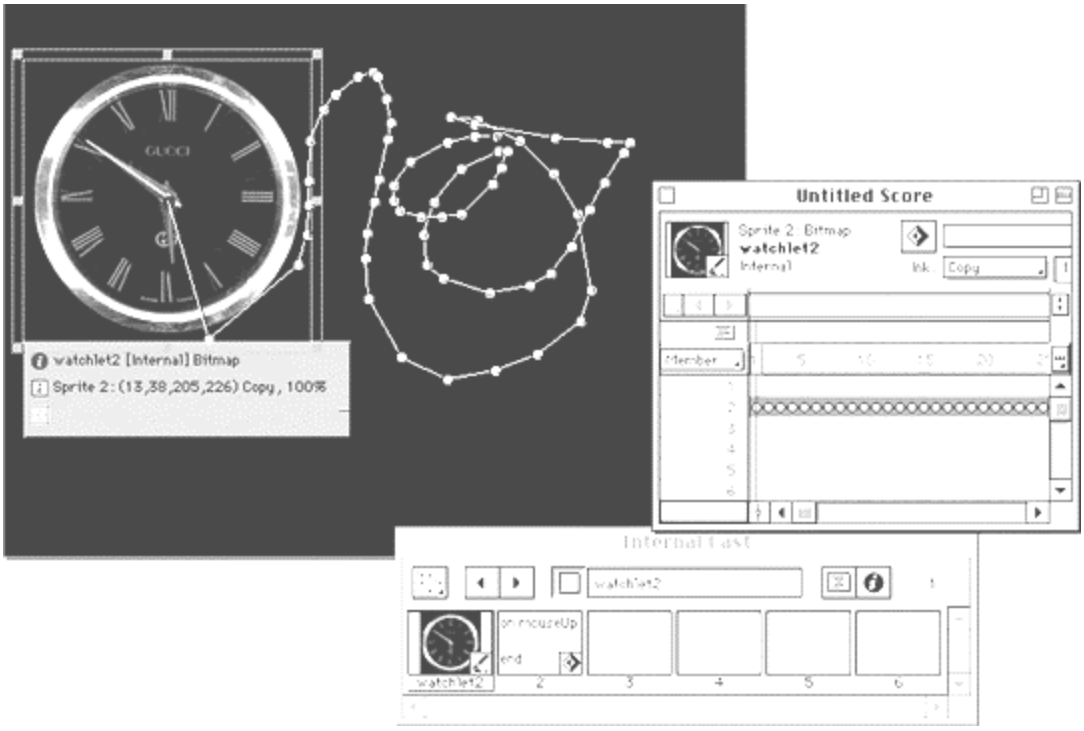
Key Frames

Key frames are recorded data in time for objects. Below we set a **Loc** or Location key frame on the first frame. We do this by pressing the **I** (I as in Insert Key) key and clicking the **Loc** selection in the popup menu. This tells Blender that on frame 1 the sphere will be located at its currently displayed point in space.



Keyframe Animation

Notice that when you used real-time animation, the animation path appeared on stage. The path is connected by a series of points. Each point represents a new keyframe. You can click on the points and drag them around to manipulate the animation paths.



Rai University
EVOLVING THINKING MINDS
Saroda Post, Dholka Taluka, District - Ahmedabad, Gujarat - 382260, India
www.raiuniversity.edu

You can also ADD key frames and REMOVE keyframes by clicking on a keyframe in the Score and selecting **INSERT > KEYFRAME** or **INSERT > REMOVE KEYFRAME** from the menu bar**. T**he more blank frames you have between keyframes, the rounder and smoother the path is between points. A linear path will connect two key frames occupying adjacent cells in the Score.

Summary

We have seen a Curves and surfaces in this unit .we start with the introduction to the topic curves and surfaces it s usage in the graphic world ,we seen polygon meshes and its representations ,next topic is parametric cubic curves and its representation, next topic is hermite curve its math calculations and basis matrix ,next topic is beizer curves its mathematics some of the example to calculate that beizer patches and the implementing the beizer curves.

The new topic in the unit is Animation its general introduction, and move to the to its tricks which include design tricks, distraction tricks, timing tricks, production tricks, implementation tricks, motion enhancement, economization tricks, etc to make the animation, next topic is introduction to the key frame animations, some examples to demonstrate all these terminology

Questions

1. what is polygon meshes?
2. How do optimize/simplify a 3D polygon mesh?
3. what is Parametric cubic curves?
4. write short notes on representation of curves:-
    1. explicit
    2. implicit
    3. and parametric
    4. How do generate a Bezier curve that is parallel to another Bezier?
6. How do split a Bezier at a specific value for t?
7. How do find a t value at a specific point on a Bezier?
8. How do fit a Bezier curve to a circle?
9. What are beizer patches?
10. What is hermite curve? Write down its basis matrix?
11. What is Animation? Explain in detail .
12. Write detail notes on various design tricks:-
    a. Attraction Tricks
    b. Distraction Tricks
    c. Timing Tricks
    d. Motion Enhancement
13. What is implementation tricks? Explain various types in brief.
14. What is Economization tricks? Explain various types in brief.
15. What is Production tricks? Explain various types in brief.
16. what is Key frame animations ?

Notes:

EIILM UNIVERSITY
SIKKIM

Jorethang, District Namchi, Sikkim- 737121, India
www.eiilmuniversity.ac.in