



CS602-Computer Graphics

(Solved MCS's)

LECTURE FROM (23 to 45)



Junaidfzal08@gmail.com
Bc190202640@vu.edu.pk

FOR MORE VISIT
VULMSHELP.COME

JUNAID MALIK
0304-1659294

AL-JUNAID TECH INSTITUTE



www.vulmshelp.com



Language Courses Training Available

I'm providing paid courses in different languages within 3 Months, Certificate will be awarded after completion.

- HTML
- CSS
- JAVASCRIPT
- BOOTSTRAPS
- IQUERY
- PHP MYSQL
- NODES.JS
- REACT JS

LMS Handling Services

LMS Activities Paid Task

Assignments 95% Results

Quizes 95% Results

GDB 95% Results

For CS619 Project Feel Free To Contact With Me

Ph# 0304-1659294
Email: junaidfzal08@gmail.com

AL-JUNAID TECH INSTITUTE

ALL answers are verified if found any mistake then Correct ACCORDINGLY

1. Computer graphics and computer vision are _____ disciplines.
 - Related
 - Interrelated
 - Same
 - **Different**
2. OpenGL has become the industry's most widely used and supported _____ graphics application programming interface (API), bringing thousands of applications to a wide variety of computer
 - platforms.
 - Dimensional
 - Dimensional
 - **2 Dimensional and 3-Dimensional**
3. _____ sets the reshape callback for the *current window*.
 - glutIdle function
 - glutKeyboardFunc
 - **glutReshapeFunc**
 - glutDisplayFunc
4. _____ sets the global idle callback to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.
 - **glutIdle function**
 - glutKeyboardFunc
 - glutReshapeFunc
 - glutDisplayFunc
5. $X^2/a^2 - y^2/b^2 = 1$ is an equation of
 - **Hyperbola Page no : 70**
 - Parabola
 - None of given
 - Ellipse

AL-JUNAID TECH INSTITUTE

6. To modify the object shape, shearing transformation cannot be used.

- False Page no : 124
- True

7. Rotation is performed around a fixed point called _____.

- Pivot
- Point
- rotation Page no : 114

8. Computer graphics is very helpful in producing graphical representations for scientific visualization and analysis

- True
- False

9. Save a line with both endpoints inside all clipping boundaries.

- Trivial Reject
- Trivial Accept Page no : 137
- None of given

10. Dark lights are nothing more than lights in which one or more of the color values are _____. Unknown

- Negative Page no : 230
- Positive Zero

11. A series of _____ computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection.

- Three Page no : 366
- Two
- Four
- Ten

AL-JUNAID TECH INSTITUTE

12. At a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our _____ cells.

- Eye Retina
- Cone Page no : 393

12. This projection technique has the direction of projection perpendicular to the viewing plane, but the viewing direction is NOT perpendicular to one of the principle faces.

Orthographic Parallel Projection

- Axonometric Parallel Projection Page no : 189
- Oblique Parallel Projection

13. The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

- Specular Page no : 234
- Ambient
- Diffuse

14. In the Phong Reflection model, _____ light is the same everywhere.

- Ambient
- Diffuse
- Specular
- Emissive

15. A plane is two dimensional since in order to uniquely define any point on its surface we require _____ numbers.

- Two
- Three
- Four
- Five

AL-JUNAID TECH INSTITUTE

16. In perspective projection, for your view to come out correctly, you will also want the _____ to pass through the middle of the screen.

- X axis
- Y axis
- **Z axis Page no : 195**
- None

17. Neither floating-point nor signed integer values are clamped to the range _____ before updating the current color.

- 0, -1.0
- -1, 1
- 1, -1

0, 1 Page no : 316

18. Bezier curve is the ideal standard for representing the _____ piecewise polynomial curves. Most complex

- Less complex
- None of given
- **More complex Page no : 333**

19. An object's _____ determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source.

- Unit
- **Normal Page no : 395**
- None of given

21. Which of the following affine transforms does NOT affect vectors?

- Scale
- Rotation
- Shear
- **Translation Page no : 113**

AL-JUNAID TECH INSTITUTE

22) This projection technique does NOT have the direction of projection perpendicular to the viewing plane.

- Orthographic Parallel Projection
- Axonometric Parallel Projection
- **Oblique Parallel Projection Page no : 189**

23) This projection technique has the direction of projection perpendicular to the viewing plane, and the viewing direction is perpendicular to one of the principle faces.

- **Orthographic Parallel Projection Page no : 189**
- Axonometric Parallel Projection
- Oblique Parallel Projection

24) In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with a `glRotatef` function call?

- **GL_MODELVIEW Page no : 388**
- GL_PROJECTION

25) In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with `glFrustum`?

- GL_MODELVIEW
- **GL_PROJECTION Page no : 369**

26) Which of the following is the order that geometry operations are performed in OpenGL (where we read the order from left to right)?

- GL_PROJECTION GL_MODELVIEW Perspective division
- **GL_MODELVIEW GL_PROJECTION Perspective division**
- Perspective division GL_PROJECTION GL_MODELVIEW
- GL_MODELVIEW Perspective division GL_PROJECTION
- GL_PROJECTION Perspective division GL_MODELVIEW

AL-JUNAID TECH INSTITUTE

27) The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Each piece of the Phong model uses different vectors and constants. Which portion does NOT include taking a dot product?

- Ambient Page no : 234
- Diffuse
- Specular

28) Which of the following is NOT a modern application for Computer Graphics---

- ► Stop-motion animation Page no : 6
- ► Computer Aided Geometric Design
- ► Video Games
- ► Scientific Visualization

29) Both Boundary Filling and Flood filling algorithms are non-recursive techniques,

- False Page no : 97
- True

30) TV series are made as simply as possible from the animation point of view. This approach is generally known as -----.

- Full animation
- Limited animation Page no : 423
- Low animation
- High resolution

31) An eight frame run cycle that ----- frame/frames to each step gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need dry brush or speed lines to make the movement flow.

- ► Two
- ► One
- ► Three

AL-JUNAID TECH INSTITUTE

- ▶ Four Page no :432

32) Reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface.

- ▶ Forward scattering
- ▶ Diffuse Lambertian
- ▶ Backscattering
- ▶ Retro Page no : 288

33) What makes this really challenging to model is that the index of refraction for most materials is a function of the ----- of the light. This means that not only is there a shift in the angle of refraction, but that the shift is different for differing----- of light.

- ▶ Reflecting angle, Reflecting angle
- ▶ Refracting angle, Refracting angle
- ▶ Frequency, Frequency
- ▶ Wavelength, Wavelength Page no : 293

34) The reflected light wave turns out to be a ----- case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Abnormal
- ▶ Complex
- ▶ Simple Page no : 291
- ▶ Unknown

35) Tessellation can be adaptive to the _____ degree of curvature of a surface.

- ▶ Local Page no : 165
- ▶ Static
- ▶ Global
- ▶ Variable

36) _____ sets the reshape call back for the *current window*. The reshape callback is triggered when a window is reshaped.

AL-JUNAID TECH INSTITUTE

- ▶ glutMainLoop
- ▶ glutIdleFunc
- ▶ glutReshapeFunc Page no : 307
- ▶ glutDisplayFunc

37) Signed integer colour components, when specified, are linearly mapped to floating-point values such that the most positive represent able value maps to 1.0, and the most negative represent able value maps to ----- . Floating-point values are mapped directly.

- ▶ -1.0
- ▶ 0.0
- ▶ 2.0
- ▶ 1.0 Page no : 315

38) NURBS stands for-----

- ▶ Non Universal Rational Binary Spline
- ▶ Non Uniform Rational Binary Splines
- ▶ Non Uniform Rational Beta Splines Page no :320
- ▶ Non Universal Rational Beta Splines

39)An orthogonal set of vectors-----

- ▶ Must be a set of linearly independent vectors
- ▶ Must be a set of linearly dependent vectors
- ▶ Must be made up of the basis vectors (e_1 , e_2 , and e_3)
- ▶ Can be made up of any set of vectors

40) Bezier curve is numerically the ----- of all the polynomial-based curves used in these applications.

- ▶ None of the given
- ▶ Most stable
- ▶ Less stable
- ▶ Most unstable

AL-JUNAID TECH INSTITUTE

40. Bezier curve is the ideal standard for representing the ----- piecewise polynomial curves.

- ▶ None of the given
- ▶ Non complex
- ▶ Most complex
- ▶ More complex repeated Question No: 15

41) Keep polygon orientations consistent to make sure that when viewed from the outside, all the polygons on the surface are oriented in the _____ direction.

- ▶ None of the given
- ▶ Neither
- ▶ Different
- ▶ Same Page no : 340

42)The-----is most simple example that exhibits the property self-similarity.

- ▶ Mosse
- ▶ Fern Page no : 350
- ▶ None of the given
- ▶ Thohar

43) A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize ----- space.

- ▶ Multi-dimensional
- ▶ One-dimensional
- ▶ Two-dimensional
- ▶ Three-dimensional Page no: 366

44)Which of the following properties of rational Bezier curves fails if the weight assigned to a control point is negative?

- ▶ End-point interpolation

AL-JUNAID TECH INSTITUTE

- ► Variation Diminishing
- ► Symmetry
- ► Convex-Hull page no : 335

45) In the Phong reflection model, there are 3 constants (a, b, c) which are used to describe the qualities of which of the following phenomena?

- ► The attenuation of a point light source with distance Page no : 285
- ► The size (in each dimension) which the light is assumed to have
- ► The amount to perturb reflection vectors as they are mirrored across the normal
- ► The material reaction to ambient, diffuse and specular light (respectively)

46) The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

- ► Specular repeated
- ► Diffuse
- ► Ambient

48) When you hit a surface in ray tracing, generally shadow rays are tested against all objects in a scene. If these rays come back saying they hit an object in the scene, which of the following do you do?

- ► add all components (i.e. ambient, diffuse and specular) from that light source to the object.
- ► add all EXCEPT the ambient light from that light source to the object (i.e. diffuse and specular)
- ► add only the ambient light from that light source to the object
- ► add none of the light from that light source to the object

49) The Color Space tool is a handy tool that we can use to interactively add two colours together to see the effects of the various strategies for handling oversaturated colours.

- ► False

AL-JUNAID TECH INSTITUTE

- ▶ True page no : 230

50) A polygon is usually defined by a sequence of ----- and Edges.

- ▶ Ending lines
- ▶ Points
- ▶ Vertices Page no : 243
- ▶ Edges

51) Which of the following properties of Bezier curves guarantees that a line passes through the control polygon as many times or more times than the line passes through the Bezier curve itself?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ Convex-Hull

52) Parity is a concept used to determine which _____ lie within a polygon. (Choose best suitable answer)

- ▶ Edge
- ▶ Vertices
- ▶ Pixels Page no : 80
- ▶ None of the given

53) The actual filling process in boundary filling algorithm begins when a point _____ of the figure is selected.

- ▶ Outside the boundary
- ▶ Inside the boundary
- ▶ At boundary
- ▶ None of the given

AL-JUNAID TECH INSTITUTE

54) Weiler-Atherton Polygon Clipping technique modify the vertex-processing procedures for window boundaries so that _____ polygons are displayed correctly.

- ► Convex
- ► Concave Page no : 245
- ► Complex
- ► None of the given

55) If a line connecting any two points within a polygon does not intersect any edge, then it will be a _____ polygon.

- ► Convex Page no : 78
- ► Concave
- ► Complex
- ► None of the given

56) _____ can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the view plane which constitutes the display surface.

- ► Mapping plane
- ► Three Coordinate Planes
- ► View plane Repeated
- ► Projection

57) The reflected light wave turns out to be a / an _____ case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ► Unknown
- ► Simple Page no: 291
- ► Complex
- ► Abnormal

58) Which of the following is NOT true about quaternions?

- They are made up of 4 numbers
- They should always be normalized to length 1
- They can be used to represent all affine transforms
- They can be used to define the rotation of an object

1) In class, we discussed three forms of shading for "Utah" graphics. Which was the first to use per vertex normals?

- Flat Shading
- Phong Shading

AL-JUNAID TECH INSTITUTE

- Gouraud Shading Page no : 240

59) Given any implicit equation, which of the following is true for all (x, y, z) that make the equation exactly zero?

- All those points are inside the object defined by the implicit equation
- All those points are on the surface of the object defined by the implicit equation Page no :205
- All those points are outside the object defined by the implicit equation
- You can't know anything without knowing what the implicit equation is

60) When solving ray-sphere intersections using the implicit equation for a sphere, you must solve the quadratic equation. Which of the following do you know if the B^2-4AC (i.e. the part under the square root) is negative?

- The ray intersects the sphere at a negative t... discard this result
- The ray intersects the sphere at a positive t... continue to the solution
- The ray does not intersect the sphere... discard this result Page no : 265
- The ray begins inside the sphere... this is a special case

61) _____ sets the global idle call back to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

- glutIdleFunc
- glutMainLoop
- glutDisplayFunc
- glutReshapeFunc

62) A space curve can be confined to a plane.

- True
- False Page no : 326

63) A tangent vector certainly defines the slope at one end of the curve, but a vector has characteristics of.....

- direction
- magnitude
- both direction and magnitude Page no : 331
- None of the given

64)

We allow the parametric variable to take on values only in the interval -----

-.

- $-1 \leq u \leq 0$
- $0 \leq u \leq 2$

AL-JUNAID TECH INSTITUTE

- $0 \leq u \leq 1$ Page no : 321

- $-1 \leq u \leq 1$

65) The degree of a Bezier curve is equal to $n-1$, where n is the number of control points

- Yes Page no : 334

- No

66) Bernstein polynomial functions are the basic functions of _____ curves.

- NURBS
- Bezier Page no : 337
- Both NURBS and Bezier
- None of the given

67)

A parametric curve is one whose defining equations are given in terms of a -----, common, independent variable called the parametric variable.

- Triple
- Double
- Single Page no : 320
- None of the given

68)

Bit mask to select a window with multisampling support. If multisampling is not available, a----- window will automatically be chosen.

- Non-multisampling Page no : 305
- Multisampling
- Mono-multisampling
- Di-multisampling

69)

Bezier curve is tangent to the lines connecting _____.

- First two points
- Last two points
- First two points and last two point
- None of the given

70) OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers-----information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

AL-JUNAID TECH INSTITUTE

- Encapsulate Page no : 297
- Shows
- Hibernates
- None of the given

71)

A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are -----.

- Binary polynomials
- Mono polynomials
- Quadratic polynomials
- Cubic polynomials Page no : 326

72) End points and an intermediate point on the curve, then we now -----
-- quantities that we can express in terms of these coefficients (3 points x 3 coordinates each), and we can use these three points to define a unique curve.

- Six
- Three
- Two
- Nine Page no : 321

73) Bezier curve can represent the more complex piecewise _____ curve.

- Polynomial Page no : 33
- Exponential
- Cubic
- None of above

74) Curve and surface equations can be expressed in either a parametric or a non parametric form.

- True
- False Page no : 333

75) Using a lighting model based upon the Blinn Phong model means that we'll always get a uniform specular highlight based upon the colour of the ----- light and material, which means that all reflections based on this model, will be reminiscent of plastic.

- Union
- Refracting Page no : 291
- Intersection
- Reflecting

76) If the current matrix (according to glMatrixMode) is multiplied by the translation matrix, with the product replacing the current matrix. That is, if M is the

AL-JUNAID TECH INSTITUTE

current matrix and T is the translation matrix, then M is replaced with -----

--.

- M-T
- Page 21 of 22
- M+T
- M/T
- **M*T**

77) With similar expressions for $y(u)$ and $z(u)$. Again the a , b , c and d terms are constant coefficients. As we did with Equation for a plane curve, we combine the $x(u)$, $y(u)$, and $z(u)$ expressions into a single vector equation $P(u) =$ _____.

- Au^2+bu^1+cu+d
- $Au^4+bu^3+cu^2+d^1$
- $Au^3+bu^2+cu^2+d$
- **Au^3+bu^2+cu+d Page no : 326**

78) The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to `gluPerspective` with a call to _____.

- `glRotated`
- `gluPerspective`
- `glTranslated`
- **`glLoadIdentity` Page no : 313**

79) Each number that makes up a matrix is called an _____ of the matrix.

- **Element Page no : 101**
- Variable
- Value
- Component

80) Which one of the following step is not involved to write pixel using video BIOS services.

- Setting desired video mode
- Using BIOS service to set color of a screen pixel
- Calling BIOS interrupt to execute the process of writing pixel.
- **Using OpenGL service to set color of a screen pixel**

81) Shadow mask methods can display a _____ range of colors.

- Small
- **Wide Page no : 29**
- Random

AL-JUNAID TECH INSTITUTE

- Crazy

82) Using Cohen-Sutherland line clipping, it is impossible for a vertex to be labeled 1111.

- True
- False

83) Intensity of the electron beam is controlled by setting _____ levels on the control grid, a metal cylinder that fits over the cathode.

- Amplitude
- Current
- Voltage Page no : 26
- Electron

84) Which of the following is NOT a modern application for Computer Graphics---

- ► Stop-motion animation (Page 6)
- ► Computer Aided Geometric Design
- ► Video Games
- ► Scientific Visualization

85) Both Boundary Filling and Flood filling algorithms are non-recursive techniques,

- ► False
- ► True

86) TV series are made as simply as possible from the animation point of view. This approach is generally known as -----.

- ► Full animation
- ► Limited animation (Page 428)
- ► Low animation
- ► High resolution

87) An eight frame run cycle that ----- frame/frames to each step gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need dry brush or speed lines to make the movement flow.

- ► Two
- ► One
- ► Three
- ► Four (Page 437)

88) reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface.

- ► Forward scattering
- ► Diffuse Lambertian

AL-JUNAID TECH INSTITUTE

- ▶ Backscattering
- ▶ Retro (Page 293)

89) What makes this really challenging to model is that the index of refraction for most materials is a function of the ----- of the light. This means that not only is there a shift in the angle of refraction, but that the shift is different for differing----- of light.

- ▶ Reflecting angle, Reflecting angle
- ▶ Refracting angle, Refracting angle
- ▶ Frequency, Frequency
- ▶ Wavelength, Wavelength (Page 229)

90) The reflected light wave turns out to be a ----- case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Abnormal
- ▶ Complex
- ▶ Simple (Page 296)
- ▶ Unknown

91) Tessellation can be adaptive to the ----- degree of curvature of a surface.

- ▶ Local (Page 170)
- ▶ Static
- ▶ Global
- ▶ Variable

92) ----- sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped.

- ▶ glutMainLoop
- ▶ glutIdleFunc
- ▶ glutReshapeFunc (Page 312)
- ▶ glutDisplayFunc

93) Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to ----- . Floating-point values are mapped directly.

- ▶ -1.0
- ▶ 0.0
- ▶ 2.0
- ▶ 1.0 (Page 320)

94) NURBS stands for ----- .

AL-JUNAID TECH INSTITUTE

- ▶ Non Universal Rational Binary Spline
- ▶ Non Uniform Rational Binary Splines
- ▶ Non Uniform Rational Beta Splines (Page 325)
- ▶ Non Universal Rational Beta Splines

95) An orthogonal set of vectors-----

- ▶ Must be a set of linearly independent vectors
- ▶ Must be a set of linearly dependent vectors
- ▶ Must be made up of the basis vectors (e_1 , e_2 , and e_3) click here 4 details
- ▶ Can be made up of any set of vectors

96) Bezier curve is numerically the ----- of all the polynomial-based curves used in these applications.

- ▶ None of the given
- ▶ Most stable (Page 338)
- ▶ Less stable
- ▶ Most unstable

97) Bezier curve is the ideal standard for representing the ----- piecewise polynomial curves.

- ▶ None of the given
- ▶ Non complex
- ▶ Most complex
- ▶ More complex (Page 338)

98) Keep polygon orientations consistent to make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction.

- ▶ None of the given
- ▶ Neither
- ▶ Different
- ▶ Same (page 345)

99) The-----is most simple example that exhibits the property self-similarity.

- ▶ Mosse
- ▶ Fern (Page 355)
- ▶ None of the given
- ▶ Thohar

100) A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize ----- space.

AL-JUNAID TECH INSTITUTE

- ▶ Multi-dimensional
- ▶ One-dimensional
- ▶ Two-dimensional
- ▶ **Three-dimensional (Page 371)**

101) Which of the following properties of rational Bezier curves fails if the weight assigned to a control point is negative?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ **Convex-Hull**

102) We want our scene to look more realistic, we should use _____ lights.

- **Ambient (Page 282)**
- Point Parallel
- Spot
- None of the given

103) This is a simple example of line clipping: the display window is the canvas and also the default-----, thus all line segments inside the canvas are drawn.

- **Clipping Rectangle (Page 141)**
- Clipping Circle
- Clipping Polygon
- Clipping Angle

104) One problem with Gouraud shading is that the----- intensities can never be greater than the intensities at the edges.

- **Triangles (Page 246)**
- Squares
- Rectangles
- Polygons

105) There is more penetration of light in case of _____ surfaces.

- Conductor (like metals)
- **Nonconductor (like dielectrics) (Page 235)**
- Both conductor and nonconductor
- None of the given

106) _____ lights should be avoided because they are not for real time environment.

- Point
- Parallel

AL-JUNAID TECH INSTITUTE

- Spot (Page 244)

- None of the given

107) The physical range of colors a device can display is called _____ of the device.

- Sharpness
- Gamut (Page 229)
- Colouring
- Colouring with Sharpness

108) _____ is simply the calculation of color reflected by the surface.

- Shading (Page 240)
- Clamping
- Scaling
- None of the given

109) When obtaining normals for a triangle, which of the following mathematical constructs is NOT used?

- Vector normalization
- Vector cross products
- Vector dot products
- Point-Point subtraction

110) Loosely, the alpha component of the RGBA quad represents the _____ of a surface.

- Opaqueness (Page 227)
- Light
- Darkness
- Shine

111) An algorithm that clips a polygon must deal with many -----cases. The case is particularly noteworthy in that the concave polygon is clipped into ----- isolate polygons.

- Similar, three
- Different, two (Page 146)
- Different, three
- Similar, two

112) _____ lighting is not dependent on any source.

- Ambient
- Diffuse
- Specular
- Emissive

113)

AL-JUNAID TECH INSTITUTE

In order to get a more realistic representation of lighting, we'll need to understand how light passes through a medium and how hitting the boundary layer at the ----- of two media can affect light's properties.

- Intersection (Page 296)
- Union
- Endpoints
- Edges

114) Lambertian shading was used mostly back when computers weren't fast enough to do _____ in real time.

- Phong shading
- Processing
- Shading
- Gouraud shading (Page 245)

115)

In Perspective Projection the point of View (POV) must lie on the _____.

- All axis
- Z axis (Page 200)
- X axis
- Y axis

116) If we want any object to glow, we should use _____ lights.

- Ambient
- Diffuse
- Specular
- Emissive (Page 240)

117) There are not many different ways of representing the intensity of a particular color element.

- True
- False (Page 276)

118) In Perspective Projection the screen plane must be parallel to the _____.

- Y-Z plane
- X-Y plane (Page 200)
- Z-Y plane
- X-Z plane

119) _____ light is reflected in all directions from surface.

- Ambient
- Diffuse (page 239)

AL-JUNAID TECH INSTITUTE

- Specular
- Emissive

120) A space curve can be confined to a plane.

- Yes
- No (Page 331)

121) To convert the information in the A matrix into that required for the P matrix, we do some simple matrix algebra, First we have $UA=UNP$ then Simply $A = \text{-----}$

- UP
- NP (Page 333)
- UN
- None

122) Perspective projection is specified with the function `glFrustum()`.

- Yes (Page 376)
- No

122) Choose a camera lens or adjust the zoom

- projection transformation (Page 372)
- viewport transformation
- modeling transformation'
- viewing transformation

123) Using a lighting model based upon the Blind Phong model means that we'll always get a uniform specular highlight based upon the color of the ----- light and material, which means that all reflections based on this model, will be reminiscent of plastic.

- Union
- Refracting
- Intersection
- Reflecting (Page 296)

124) Refractive index is a function of temperature, mostly due to density changes in materials with changes in temperature.

- True (Page 300)
- False

125)

Length L depends on the angle alpha and the z coordinate of the point to be projected and L can be represented by -----.

- $z * 1 / \tan(\alpha)$ (Page 198)
- $z * L^2$
- $z * 1 / \tan(\beta)$

AL-JUNAID TECH INSTITUTE

- $z * 1 / \tan (\text{gamma})$

126) The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the _____ light.

- Ambient
- Ambient, diffuse, and specular (Page 281)
- Specular
- Diffuse, and specular

127)

Another way to define a space curve by using intermediate points and the tangents at each end for making the curve

- Yes
- No (Page 334)

128) An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, ----- graphics standard.

- Tertiary
- Binary
- Single platform
- Multiplatform (Page 301)

129) `glutReshapeWindow` requests a change in the size of the current window. The width and height parameters are size extents in pixels. The width and height must be ___ values.

- Neutral
- Negative
- Positive (Page 311)
- None of the given

130) A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are -----.

- Binary polynomials
- Mono polynomials
- Quadratic polynomials
- Cubic polynomials (Page 331)

131)

Refractive index is a function of temperature, mostly due to changes in ----- of materials with changes in temperature. A simple correction can be applied in most circumstances to allow us to use a value given at one temperature at another.

- Density (Page 300)

AL-JUNAID TECH INSTITUTE

- pressure
- nature
- volume

132) If we assign a different value to the parametric variable for the intermediate point, then we obtain different values for the coefficients. This, in turn, means that a different curve is produced, although it passes through the----- three points.

- isolate
- different
- same (Page 328)
- none

133) The attenuation formula is $f = \frac{1}{C + Ld + Qd^2}$, where C, L and Q are the constant, linear and quadratic attenuation factors and d is the distance between the vertex being lit and the light source.

- $\frac{1}{C + Ld + Qd^2}$
- $\frac{1}{C + Ld + Qd}$
- $\frac{1}{C + L + d + Qd^2}$
- $\frac{1}{Cd + Ld + Qd^2}$

144) Bezier curve is tangent to the lines connecting _____.

- First two points
- Last two points
- First two points and last two point (Page 340)
- None of the given

145) End points and an intermediate point on the curve, then we now ----- quantities that we can express in terms of these coefficients (3 points x 3 coordinates each), and we can use these three points to define a unique curve.

- Six
- Three
- Two
- Nine (Page 326)

146) Choose a camera lens or adjust the zoom

- projection transformation (Page 372)
- viewport transformation
- modeling transformation
- viewing transformation

148) _____ OpenGL function is used for aiming and positioning the camera towards the object

- `glLoadIdentity()` (Page 375)

AL-JUNAID TECH INSTITUTE

- gluLookAt()
- glFrustum()
- None of Above

149) A parametric curve is one whose defining equations are given in terms of a ---
-----, common, independent variable called the parametric variable.

- Triple
- Double
- **Single (Page 325)**
- None of the given

150) The reflection coefficients are in the----- range and are specified as part of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient.

- [0, 10]
- **[0, 1] (Page 281)**
- [0, 5]
- [0, 2]

151) To ensure a smooth transition from one section of a piecewise _____ to the next, we can impose various continuity conditions at the connection points on parametric curve parametric curve

- **polygon vector (not confirm) (Page 245)**
- None of the these

152) The curve is always contained within the _____ of the control points

- Tangents
- **Convex Hull (Page 340)**
- Subdivision
- None of Above

Question # 1

A space curve can be confined to a plane.

- Yes
- **No (Page 331)**

Question # 2

To convert the information in the A matrix into that required for the P matrix, we do some simple matrix

algebra, First we have $UA=UNP$ then Simply $A =$ -----

- UP
- **NP (Page 333)**

AL-JUNAID TECH INSTITUTE

- UN
- None
- 16

Question #3

Perspective projection is specified with the function glFrustum().

- Yes (Page 376)
- No

Question # 4

Choose a camera lens or adjust the zoom

- projection transformation (Page 372)
- viewport transformation
- modeling transformation'
- viewing transformation

Question # 5

Using a lighting model based upon the Blinn Phong model means that we'll always get a uniform specular highlight based upon the color of the ----- light and material, which means that all reflections based on this model, will be reminiscent of plastic.

- Union
- Refracting
- Intersection
- Reflecting (Page 296)

Question # 6

Refractive index is a function of temperature, mostly due to density changes in materials with changes in temperature.

- True (Page 300)
- False

Question # 7

Length L depends on the angle alpha and the z coordinate of the point to be projected and L can be represented by-----.

- $z * 1 / \tan(\alpha)$ (Page 198)
- $z * L^2$
- $z * 1 / \tan(\beta)$
- $z * 1 / \tan(\gamma)$

- 17

Question # 8

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the _____ light.

AL-JUNAID TECH INSTITUTE

- Ambient
- Ambient, diffuse, and specular (Page 281)
- Specular
- Diffuse, and specular

Question # 9

Another way to define a space curve by using intermediate points and the tangents at each end for making the curve

- Yes
- No (Page 334)

Question # 10

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, ----- graphics standard.

- Tertiary
- Binary
- Single platform
- Multiplatform (Page 301)

Question # 11

glutReshapeWindow requests a change in the size of the current window. The width and height parameters are size extents in pixels. The width and height must be _____ values.

- Neutral
- Negative
- Positive (Page 311)
- None of the given

Question # 12

A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are -----.

- Binary polynomials
- Mono polynomials
- Quadratic polynomials
- Cubic polynomials (Page 331)

Question # 13

Refractive index is a function of temperature, mostly due to changes in ----- of materials with changes in temperature. A simple correction can be applied in most circumstances to allow us to use a value given at one temperature at another.

- Density (Page 300)

AL-JUNAID TECH INSTITUTE

- pressure
- nature
- volume

Question # 14

If we assign a different value to the parametric variable for the intermediate point, then we obtain different values for the coefficients. This, in turn, means that a different curve is produced, although it passes through the ----- three points.

- isolate
- different
- same (Page 328)
- none

Question #15

Bezier curve is tangent to the lines connecting _____.

- First two points
- Last two points
- First two points and last two point (Page 340)
- None of the given

Question # 16

End points and an intermediate point on the curve, then we now ----- quantities that we can express in terms of these coefficients (3 points x 3 coordinates each), and we can use these three points to define a unique curve.

- Six
- Three
- Two
- Nine (Page 326)
- 19

Question # 17

Choose a camera lens or adjust the zoom

- projection transformation (Page 372)
- viewport transformation
- modeling transformation
- viewing transformation

QuestionZ#18

_____ OpenGL function is used for aiming and positioning the camera towards the object

- glLoadIdentity() (Page 375)

AL-JUNAID TECH INSTITUTE

- gluLookAt()
- glFrustum()
- None of Above

Question # 19

A parametric curve is one whose defining equations are given in terms of a -----, common, independent variable called the parametric variable.

- Triple
- Double
- **Single (Page 325)**
- None of the given

Question # 20

The reflection coefficients are in the-----range and are specified as part of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient.

- [0, 10]
- **[0, 1] (Page 281)**
- [0, 5]
- [0, 2]

Question # 21

To ensure a smooth transition from one section of a piecewise_____to the next, we can impose various continuity conditions at the connection points

- non parametric curve
- parametric curve
- **polygon vector (not confirm) (Page 245)**
- None of the these

Question # 22

The curve is always contained within the_____of the control points

- Tangents
- **Convex Hull (Page 340)**
- Subdivision
- None of Above

Question # 23

Projection can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the-----, which constitutes the display surface. The mapping is determined by a projection line called the projector that passes through P and intersects the -----.

- Two Coordinate Planes

AL-JUNAID TECH INSTITUTE

- View plane or projection plan (Page 193)
- Three Coordinate Planes
- Mapping plane

Question # 24

Determine how large we want the final photograph to be - for example, we might want it enlarged

- projection transformation
- viewport transformation (Page 372)
- modeling transformation
- viewing transformation

Question # 25

Ambient light is the light that comes from -----directions, thus all surfaces are illuminated equally regardless of orientation. However, this is a big hack in traditional lighting calculations since "real" ambient light really comes from the light reflected from the "environment."

- All (Page 281)
- Opposite
- Same
- Four different

Question # 26

Silhouette edges occur when dot product of surface normal vector and the view vector is _____.

- Zero (Page 345)
- One
- Both zero and one

Question # 27

If the current matrix (according to `glMatrixMode`) is multiplied by the translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with-----.

- $M-T$
- $M+T$
- M/T
- $M*T$ (Page 317)

Question # 28

Arrange the scene to be photographed into the desired composition

- projection transformation
- viewport transformation
- modeling transformation (Page 317)

AL-JUNAID TECH INSTITUTE

- viewing transformation

Question # 29

In the forms of texture mapping, Image to world space and world space to image, each suffers from different problems related to magnifications and magnification. Which of the two shows the following problem: When the texture is larger than the screen space it maps to, many texture units (texels) are never sampled?

- Image to world space
- World space to image
- X-axis
- Y-axis

Question # 31

Imagine a curve in three-dimensional space, each point on the curve has a unique set of coordinates: a specific x value, y value, and z value. Each coordinate is controlled by a ----- parametric equation.

- Opposite
- Similar
- Separate (Page 325)

Question # 32

We allow the parametric variable to take on values only in the interval -----

-.

- $-1 \leq u \leq 0$
- $0 \leq u \leq 2$
- $0 \leq u \leq 1$ (Page 326)
- $-1 \leq u \leq 1$

Question # 33

Bezier curve can represent the more complex piecewise _____ curve.

- Polynomial (Page 338)
- Exponential
- Cubic
- None of above

Question # 34

A fractal is generally a property called _____.

- Fractal Dimension
- Self-similarity (Page 355)
- Koch Curve
- None of above

Question # 35

Normalized cross product of two vectors on that surface provides normal vector

AL-JUNAID TECH INSTITUTE

- Yes (Page 347)
- No

Question # 36

Every point on a curve has a straight line associated with it called the _____
State line

- tangent line (Page 334)
- curved line
- None of the given

Question # 36

The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at ----- . This window identifier can be used when calling glutSetWindow.

- Three
- Two
- One (Page 308)
- Zero

Question # 37

Curve and surface equations can be expressed in either a parametric or a non parametric form.

- True
- False

Question # 38

Bernstein polynomial functions are the basic functions of _____ curves.

- NURBS
- Bezier (Page 342)
- Both NURBS and Bazier
- None of the given

Question # 39

Geometric patterns that is repeated at ever smaller scales to produce irregular shapes and surfaces are called _____

- Geometric patterns
- Fractals (Page 352)
- Animated components
- Segments

Question # 40

In order to get a more realistic representation of lighting, we'll need to understand how light passes through a medium and how hitting the boundary layer at the -----
----- of two media can affect light's properties.

AL-JUNAID TECH INSTITUTE

- **Intersection (Page 296)**
- Union
- Endpoints
- Edges

Question # 41

_____ sets the global idle call back to be 'func' so a GLUT program can perform backgroundprocessing tasks or continuous animation when window system events are not being received.

- **glutIdleFunc (Page 313)**
- glutMainLoop
- glutDisplayFunc
- glutReshapeFunc

Question # 42

A tangent vector certainly defines the slope at one end of the curve, but a vector has characteristics of.....

- direction
- magnitude
- **both direction and magnitude (Page 336)**
- None of the given

Question # 43

The degree of a Bezier curve is equal to $n-1$, where n is the number of control points

- Yes (Page 339)
- No

Question # 44

Bit mask to select a window with multisampling support. If multisampling is not available, a----- window will automatically be chosen.

- **Non-multisampling (Page 310)**
- Multisampling
- Mono-multisampling
- Di-multisampling

Question # 45

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers -----information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

AL-JUNAID TECH INSTITUTE

- Encapsulate (Page 302)
- Shows
- Hibernates
- None of the given

Question # 46

With similar expressions for $y(u)$ and $z(u)$. Again the a , b , c and d terms are constant coefficients. As we did with Equation for a plane curve, we combine the $x(u)$, $y(u)$, and $z(u)$ expressions into a single vector equation

$P(u) = \underline{\hspace{2cm}}$.

- $Au^2+bu+cu+d$
- $Au^4+bu^3+cu^2+d$
- Au^3+bu^2+cu+d
- Au^3+bu^2+cu+d (Page 331)

Question # 48

The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto the current matrix, stack instead, precede the call to `gluPerspective` with a call to -----.

- `glRotated`
- `gluPerspective` (Page 318)
- `glTranslated`
- `glLoadIdentity`

Question # 49

The basic functions $f_i(u)$ in Bezier curve must be symmetric with respect to u and $(u-2)$

- yes
- no (Page 341)

Question # 50

Arrange the scene to be photographed into the desired composition

- projection transformation
- viewport transformation
- modeling transformation (Page 372)
- viewing transformation

Question No: 51

NURBS stands for-----.

AL-JUNAID TECH INSTITUTE

- Non Universal Rational Binary Spline
- Non Uniform Rational Binary Splines
- **Non Uniform Rational Beta Splines (Page 325)**
- Non Universal Rational Beta Spline
- Question No: 1
- Which of the following is NOT a modern application for Computer Graphics_____

- **▶ Stop-motion animation (Page 6)**
- ▶ Computer Aided Geometric Design
- ▶ Video Games
- ▶ Scientific Visualization

- Question No: 52
- Both Boundary Filling and Flood filling algorithms are non-recursive techniques,
- **▶ False**
- ▶ True

Question No: 53

TV series are made as simply as possible from the animation point of view. This approach is generally known as -----.

- ▶ Full animation
- **▶ Limited animation (Page 431)**
- ▶ Low animation
- ▶ High resolution

Question No: 54

An eight frame run cycle that ----- frame/frames to each step gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need dry brush or speed lines to make the movement flow.

- ▶ Two
- ▶ One

AL-JUNAID TECH INSTITUTE

▶ Three

▶ Four (Page 440)

Question No: 55

----- Reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface.

- ▶ Forward scattering
- ▶ Diffuse Lambertian
- ▶ Backscattering
- ▶ Retro (Page 296)

Question No: 56

What makes this really challenging to model is that the index of refraction for most materials is a function of the ----- of the light. This means that not only is there a shift in the angle of refraction, but that the shift is different for differing ----- of light.

- ▶ Reflecting angle, Reflecting angle
- ▶ Refracting angle, Refracting angle
- ▶ Frequency, Frequency
- ▶ Wavelength, Wavelength (Page 301)

Question No: 57

The reflected light wave turns out to be a ----- case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Abnormal
- ▶ Complex
- ▶ Simple (Page 299)
- ▶ Unknown

Question No: 58

Tessellation can be adaptive to the _____ degree of curvature of a surface.

AL-JUNAID TECH INSTITUTE

- ▶ Local (Page 173)
- ▶ Static
- ▶ Global
- ▶ Variable

Question No: 59

_____ sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped.

- ▶ glutMainLoop
- ▶ glutIdleFunc
- ▶ glutReshapeFunc (Page 315)
- ▶ glutDisplayFunc

Question No: 60

Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to ----- --. Floating-point values are mapped directly.

- ▶ -1.0
- ▶ 0.0
- ▶ 2.0
- ▶ 1.0 (Page 323)

Question No: 61

NURBS stands for-----.

- ▶ Non Universal Rational Binary Spline
- ▶ Non Uniform Rational Binary Splines
- ▶ Non Uniform Rational Beta Splines (Page 328)
- ▶ Non Universal Rational Beta Splines

Question No: 62

An orthogonal set of vectors-----

AL-JUNAID TECH INSTITUTE

- ▶ Must be a set of linearly independent vectors
- ▶ Must be a set of linearly dependent vectors
- ▶ **Must be made up of the basis vectors (page no 254)**
- ▶ Can be made up of any set of vectors

Question No: 63

Bezier curve is numerically the-----of all the polynomial-based curves used in these applications.

- ▶ None of the given
- ▶ **Most stable (Page 341)**
- ▶ Less stable
- ▶ Most unstable

Question No: 64

Bezier curve is the ideal standard for representing the ----- piecewise polynomial curves.

- ▶ None of the given
- ▶ Non complex
- ▶ Most complex
- ▶ **More complex(Page 341)**

Question No: 65

- Keep polygon orientations consistent to make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction.
 - ▶ None of the given
 - ▶ Neither
 - ▶ Different
 - ▶ **Same (page 347)**

Question No: 66

The ----- is simplest example that exhibits the property self-similarity.

- ▶ Moses

AL-JUNAID TECH INSTITUTE

- ▶ Fern (Page 358)
- ▶ None of the given
- ▶ Thohar

Question No: 66

A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize ---
_____ space.

- ▶ Multi-dimensional
- ▶ One-dimensional
- ▶ Two-dimensional
- ▶ Three-dimensional (Page 374)

Question No: 68

Which of the following properties of rational Bezier curves fails if the weight assigned to a control point is negative?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ Convex-Hull

Question No: 69

Actually, each component is a rational Bézier curve. We have made it very clear that all weights must be non-negative. If some of them are negative, the strong convex hull property or even the convex hull property will not hold.

Question No: 70

In the Phone reflection model, there are 3 constants (a, b, c) which are used to describe the qualities of which of the following phenomena?

- ▶ The attenuation of a point light source with distance
- ▶ The size (in each dimension) which the light is assumed to have
- ▶ The amount to perturb reflection vectors as they are mirrored across

AL-JUNAID TECH INSTITUTE

the normal

- ▶ The material reaction to ambient, diffuse and specular light (respectively)

Question No: 71

The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

- ▶ **Specular**
- ▶ Diffuse
- ▶ Ambient

Question No: 72

When you hit a surface in ray tracing, generally shadow rays are tested against all objects in a scene. If these rays come back saying they hit an object in the scene, which of the following do you do?

- ▶ add all components (i.e. ambient, diffuse and specular) from that light source to the object.
- ▶ add all EXCEPT the ambient light from that light source to the object (i.e. diffuse and specular)
- ▶ **add only the ambient light from that light source to the object**
- ▶ add none of the light from that light source to the object

Question No: 73

The Color Space tool is a handy tool that we can use to interactively add two colours together to see the effects of the various strategies for handling oversaturated colours.

- ▶ False
- ▶ **True (Page 238)**

Question No: 74

A polygon is usually defined by a sequence of-----and Edges.

AL-JUNAID TECH INSTITUTE

- ▶ Ending lines
- ▶ Points
- ▶ Vertices (Page 149)
- ▶ Edges

Question No: 75

Which of the following properties of Bezier curves guarantees that a line passes through the control polygon as many times or more times than the line passes through the Bezier curve itself?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ Convex-Hull

Question No: 76

Parity is a concept used to determine which _____ lie within a polygon.
(Choose best suitable answer)

- ▶ Edge
- ▶ Vertices
- ▶ Pixel (Page 80)
- ▶ None of the given

Question No: 77

The actual filling process in boundary filling algorithm begins when a point _____ of the figure is selected.

- ▶ Outside the boundary
- ▶ Inside the boundary (Page 105)
- ▶ At boundary
- ▶ None of the given

Question No: 78

Clipping technique modifies the vertex-processing procedures for window boundaries so that _____ polygons are displayed correctly.

AL-JUNAID TECH INSTITUTE

- ▶ Convex
- ▶ Concave (Page 158)
- ▶ Complex
- ▶ None of the given

Question No: 79

If a line connecting any two points within a polygon does not intersect any edge, then it will be a _____ polygon.

- ▶ Convex (Page 79)
- ▶ Concave
- ▶ Complex
- ▶ None of the given

Question No: 80

_____ can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the view plane which constitutes the displaysurface.

- ▶ Mapping plane
- ▶ Three Coordinate Planes
- ▶ View plane
- ▶ Projection (Page 265)

Question No: 81

The reflected light wave turns out to be a / an _____ case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Unknown
- ▶ Simple (Page 299)
- ▶ Complex
- ▶ Abnormal

Question No: 82

Computer graphics and computer vision are _____ disciplines.

AL-JUNAID TECH INSTITUTE

- Related (Page 24)
- Interrelated
- Same
- Different

Question No: 83

OpenGL has become the industry's most widely used and supported _____ graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms.

- 2-Dimensional
- 3-Dimensional
- 2-Dimensional and 3-Dimensional (Page 301)

Question No: 84

----- sets the global idle callback to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

- glutIdle function (Page 313)
- glutKeyboardFunc
- glutReshapeFunc
- glutDisplayFunc

Question No: 85

$X^2/a^2 - y^2/b^2 = 1$ is an equation of

- Hyperbola (Page 70)
- Parabola ($4px=y^2$)
- None of given
- Ellipse ($X^2/a^2 + y^2/b^2 = 1$)

Question No: 86

To modify the object shape, shearing transformation cannot be used.

AL-JUNAID TECH INSTITUTE

- False (Page 195)
- True

Question No: 87

Rotation is performed around a fixed point called_____.

- Pivot point rotation (Page 122)

Question No: 88

Computer graphics is very helpful in producing graphical representations for scientific visualization and analysis

- True (Page 9)
- False

Question No: 89

Save a line with both endpoints inside all clipping boundaries.

- Trivial Reject
- Trivial Accept (Page 145)
- None of given

Question No: 90

Dark lights are nothing more than lights in which one or more of the color values are_____.

- Unknown
- Negative (Page 238)
- Positive
- Zero

Question No: 91

A series of_____ computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection

AL-JUNAID TECH INSTITUTE

operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection.

- Three (Page 374)
- Two
- Four
- Ten

Question No: 92

At a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our _____ cells.

- Eye
- Retina
- Cone (Page 401)

Question No: 93

- In the Phong Reflection model, _____ light is the same everywhere.
- Ambient
- Diffuse
- Specular
- Emissive

Question No: 94

This projection technique has the direction of projection perpendicular to the viewing plane, but the viewing direction is NOT perpendicular to one of the principle faces.

- Orthographic Parallel Projection (Page 197)
- Axonometric Parallel Projection
- Oblique Parallel Projection

Question No: 95

A plane is two dimensional since in order to uniquely define any point on its surface we require _____ numbers.

AL-JUNAID TECH INSTITUTE

- Two (Page 359)
- Three
- Four

Question No: 96

In perspective projection, for your view to come out correctly, you will also want the _____ to pass through the middle of the screen.

- X axis
- Y axis
- Z axis (Page 203)
- None

Question No: 97

Neither floating-point nor signed integer values are clamped to the range _____ before updating the current color.

- 0 , -1.0
- -1 , 1
- 1 , -1
- 0, 1 (Page 324)

Question No: 98

An object's _____ determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source.

- Unit
- Normal (Page 403)
- None of given

Question No: 99

Which was the oldest shading model?

AL-JUNAID TECH INSTITUTE

- Flat Shading (Page 249)
- Phong Shading
- Gouraud Shading

Question No: 100

Which of the following affine transforms does NOT affect vectors?

- Scale
- Rotation
- Shear
- Translation

Question No: 101

We want our scene to look more realistic, we should use _____ lights.

- Ambient (Page 285)
- Point Parallel
- Spot
- None of the given

Question No: 102

This is a simple example of line clipping: the display window is the canvas and also the default -----, thus all line segments inside the canvas are drawn.

- Clipping Rectangle (Page 144)
- Clipping Circle
- Clipping Polygon
- Clipping Angle

Question No: 103

One problem with Gouraud shading is that the -----intensities can never be greater than the intensities at the edges.

- Triangles (Page 249)

AL-JUNAID TECH INSTITUTE

- Squares
- Rectangles
- Polygons

Question No: 104

There is more penetration of light in case of _____ surfaces.

- Conductor (like metals)
- Nonconductor (like dielectrics) (Page 243)
- Both conductor and nonconductor
- None of the given

Question No: 105

_____ Lights should be avoided because they are not for real time environment.

- Point
- Parallel
- Spot (Page 247)
- None of the given

Question No: 106

The physical range of colors a device can display is called _____ of the device.

- Sharpness
- Gamut (Page 232)
- Colouring
- Colouring with Sharpness

Question No: 107

_____ is simply the calculation of color reflected by the surface.

- Shading (Page 243)
- Clamping

AL-JUNAID TECH INSTITUTE

- Scaling
- None of the given

Question No: 108

When obtaining normal for a triangle, which of the following mathematical constructs is NOT used?

- Vector normalization
- Vector cross products
- Vector dot products
- Point-Point subtraction

Question No: 109

Loosely, the alpha component of the RGBA quad represents the _____ of a surface.

- Opaqueness (Page 239)
- Light
- Darkness
- Shine

Question No: 110

An algorithm that clips a polygon must deal with many ----- cases. The case is particularly note worthy in that the concave polygon is clipped into -----
- isolate polygons

- Similar, three
- Different, two (Page 146)
- Different, three
- Similar, two

Question No: 111

The circle and ellipse are symmetric across 8 octants.

- True

AL-JUNAID TECH INSTITUTE

- False (Page 60)

Question No: 112

UV light is used in Plasma Panel displays to excite phosphor.

- True
- False

Question No: 113

Which of the following is not true about matrices?

- $A + B = B + A$
- $a(A + B) = aA + aB$
- $(AT)T = AT$
- $A + (B + C) = (A + B) + C$

Question No: 114

According to Odd Parity Rule, a point is inside the polygon, if:

- Line from an outside point to this point does not cross the edges odd number of times
- Line from any point to this point crosses the edges odd number of times
- Line from an outside point to this point crosses the edges odd number of times (Page 80)
- Line from this point to any point outside the polygon intersects any edge

Question No: 115

As opposed to direct memory access method, BIOS routines provide an easier and faster method of drawing pixels on screen.

- True
- False (Page 47)

Question No: 116

AL-JUNAID TECH INSTITUTE

When a point P(x,y) is rotated by θ the coordinates of transformed point P' are given as:

- $x' = x \cos(\theta) - y \sin(\theta), y' = x \sin(\theta) + y \cos(\theta)$ (Page 181)
- $x' = y \cos(\theta) - x \sin(\theta), y' = y \sin(\theta) + x \cos(\theta)$
- $x' = x \cos(\theta) + y \sin(\theta), y' = x \sin(\theta) - y \cos(\theta)$
- $x' = y \cos(\theta) + x \sin(\theta), y' = y \sin(\theta) - x \cos(\theta)$

Question No: 117

The amount of time it takes to illuminate a specific location on phosphor coated screen is called Persistence.

- True
- False (Page 27)

Question No: 118

Incremental line drawing algorithm makes use of the equation of straight line.

- True
- False (page 54)

Question No: 119

In matrix multiplication:

- The two matrices must be square
- The number of rows of 1st matrix must be the same as the number of columns of the second.
- The two matrices must either be row matrices or column matrices
- The number of columns of 1st matrix must be the same as the number of rows of the second.

Question No: 120

In Horizontal retrace, after completion of all the pixels in a scan line, the refreshing continues from the 1st pixel of the next scan line.

- True

AL-JUNAID TECH INSTITUTE

- False(Page 28)

Question No: 121

Parity Rule is used to determine whether a pixel is inside a polygon or not.

- True (Page 80)
- False

Question No: 122

When dot product of two vectors equals zero, this implies that the two vectors are:

- parallel to each other
- orthogonal (perpendicular) to each other. (Page 177)
- intersect each other
- equal to each other

Question No: 123

In Pixmap exactly one bit is used to hold color value of each pixel.

- True
- False (Page 28)

Question No: 124

The Boundary Fill and Flood Fill algorithms:

- Must use 4-connected approach
- Must use 8-connected approach
- May use 4-connected or 8-connected approach
- Must not use recursive approach

Question No: 125

To show 256 colors, the no of bits required for each pixel are

- 8 (Page 39)
- 16
- 32

AL-JUNAID TECH INSTITUTE

- 64

Question No: 126

25 * 80 resolution with 16 colors supports

- **Text mode (Page 43)**
- Graphics mode
- Both
- None

Question No: 127

Two matrices are said to be equal, if they have

- same order
- same corresponding elements
- **Same order and same corresponding elements.(page 11)**
- Different elements.

Question No: 128

Two points are said to be collinear, if they lie on the

- **same line (page 53)**
- different but parallel lines
- either on the same plane or two parallel planes
- different plane

Question No: 129

A Polygon is convex, if the line connecting:

- Any two points outside the polygon intersects its boundary
- **Any two points inside the polygon don't intersect any edges of the polygon. (Page 171)**
- A point inside the boundary with any point outside, does not intersect the polygon boundary
- Any two vertices, intersects some edge of polygon.

AL-JUNAID TECH INSTITUTE

Question No: 130

Polygon is complex, if the line connecting:

- Any two points outside the polygon intersects its boundary.
- Any two points inside the polygon intersects its boundary.
- A point inside the boundary with any point outside does not intersect the polygon boundary.
- Any two vertices, intersects some edge of polygon. (Page 79)

Question No: 131

The equation of hyperbola centered at origin (if the transverse axis is along x -axis) can be given as:

- $x^2/b^2 - y^2/a^2 - 1 = 0$
- $x^2/b^2 + y^2/a^2 + 1 = 0$
- $x^2/a^2 - y^2/b^2 - 1 = 0$
- $x^2/b^2 - y^2/a^2 - 1 = 0$

Question No: 132

Which one is not valid out code to perform trivial accept / reject test in line clipping:

- 1101
- 1001 (Page 143)
- 0101
- 0110

Question No: 133

Which one of the following is not the graphics library is use:

- FastGL
- OpenGL
- DirectX
- EasyGL (Page 47)

AL-JUNAID TECH INSTITUTE

Question No: 134

In video text memory, _____ are used to display a character.

- 2 bytes (Page 43)
- 4 bytes
- 8 bytes
- 16 bytes

Question No: 135

In _____ algorithm(s), old color must be read before it is invoked.

- Scan line filling
- Flood filling (Page 104)
- Both scan line and flood filling
- Scan filling

Question No: 136

According to the architecture of raster graphics system, display processor memory will act as _____.

- Video controller (Page 36)
- System memory
- Frame buffer
- Video controller and System memory

Question No: 137

Global coordinate systems can be defined with respect to local coordinate system

- True
- False (Page 258)

Question No: 138

Cross product of two vectors results in a _____.

- Magnitude

AL-JUNAID TECH INSTITUTE

- Vector (Page 117) cross product of 2 vectors is a vector.
- Scalar
- Value

Question No: 139

Shortcoming of Sutherland-Hodgeman Algorithm is concave polygons may be displayed with extraneous lines

- True (Page 158)
- False

Question No: 140

In 2D transformations, rotations applied to a point P can be denoted as _____ (Where? Represents theta).

- $P' = R(\theta_1 + \theta_2) \cdot P$ (Page 127)
- $P' = (R(\theta_1) + R(\theta_2)) \cdot P$
- $P' = R(\theta_1 \times \theta_2) \cdot P$
- $P' = R(\theta_1) \cdot P$

Question No: 141

A straight line can be moved to another location by applying _____ to each of the line endpoints and redrawing the line between the new coordinates.

- Rotation
- Translation (Page 121)
- Reflection
- Scaling factor

Question No: 142

Locations can be translated or "transformed" from one coordinate system to the other.

- True (Page 166)
- False

AL-JUNAID TECH INSTITUTE

Question No: 143

If the value of scaling factors s_x and s_y is greater than 1, then size of objects will be _____.

- Reduced
- Enlarged (Page 121)
- Remain same
- Shear

Question No: 144

A column matrix is also known as _____.

- Column vector (Page 110)
- Row vector
- Vector
- Simple Matrix

Question No: 145

Dot product of two vectors results in _____ quantity.

- Scalar (Page 118)
- Vectors
- Magnitude
- Value

Question No: 146

Set up your tripod and pointing the camera at the scene

- projection transformation
- viewport transformation
- modeling transformation
- viewing transformation (Page 375)

Question No: 147

_____ is based on characteristic size or scale

AL-JUNAID TECH INSTITUTE

- Fractal Geometry
- Traditional Geometry
- **Euclidean Geometry (Page 362)**
- None of Above

Question No: 148

Bernstein polynomial functions are the basic functions of _____ curves.

- NURBS
- **Bezier (Page 342)**
- Both NURBS and Bazier
- None of the given

Question No: 149

Which of the following does NOT figure into the Field of View of a pinhole camera?

- **The direction of projection**
- The distance from the center of projection to the projection plane
- The size of the projection plane

Question No: 150

When transforming a random Axis-Aligned Bounding Box defined by the points (nearx, neary, nearz) and (farx, fary, farz) to the standard orthographic viewing box, which affine transforms are used?

- shear and translation
- rotation and scale
- scale and shear
- **translation and scale**

Question No: 150

In class, we discussed the purpose of the front and back clipping planes in OpenGL. Which of the following was NOT a purpose for using clipping planes?

AL-JUNAID TECH INSTITUTE

- division by zero
- objects behind the center of projection mapping onto the projection plane
- **avoiding the problems of infinite viewing volume size**

Question No: 152

In “Utah” graphics, lights are simplified in order to approximate light/matter interaction with a minimum amount of work. Which of the following is NOT true about the simplifications made in “Utah” graphics lights?

- Light intensity and color are folded into one value.
- Lights are assumed to have zero size
- **Spotlights cannot be handled**
- Soft shadows cannot be handled

Question No: 153

In the Phone Reflection model, ambient light is the same everywhere.

- **true**
- false

Question No: 154

We discussed several global illumination algorithms in class. Which of the following listed can properly handle diffuse-diffuse reflections

- Ray Tracing
- **Radiosity**
- Photon Mapping
- RenderMan

Question No: 155

We discussed several global illumination algorithms in class. Which of the following listed can properly handle caustics?

- Ray Tracing
- Radiosity

AL-JUNAID TECH INSTITUTE

- Photon Mapping
- RenderMan

Question No: 156

In radiosity a matrix is formed of size $n \times n$. Which of the following can be known if the (i, j) position in the matrix is zero?

- patch i is much larger than patch j
- patch i is much smaller than patch j
- patch i is visible from patch j
- patch i is not visible from patch j

Question No: 157

When solving for ray-polygon intersections, after intersecting the ray with a plane, the dominant component of the plane normal is found. this is used to

- ignore any component other than the dominant when you project to 2D
- ignore the dominant component when you project to 2D
- solve the inside-outside test only for that component

Question No: 158

If solving for inside-outside of a triangle using barycentric coordinates, and you have two barycentric coordinates solved ($bc_1 = .57$, $bc_2 = .62$) which of the following do you know?

- The point is outside the triangle
- The point is on the boundary of the triangle
- The point is inside the triangle
- The hit point is on the "back face" of the triangle

Question No: 159

The majority of the execution time of a ray tracer is spent in ray-object intersection code.

- true

AL-JUNAID TECH INSTITUTE

- false

Question No: 160

In the Pixar short “Geri’s Game”, the trees in the background were created using which of the following techniques?

- **Fractals**
- Bump mapping
- Environment mapping
- Catmull-Clark Subdivision Surfaces

Question No: 161

The basic functions $f_i(u)$ in Bezier curve must be symmetric with respect to u and $(u-2)$

- yes
- **no (Page 344)**

Question No: 162

In the Pixar short “Geri’s Game”, Geri’s glasses seemed to bend the light as it passed through. Which of the following techniques was used?

- Fractals
- Bump mapping
- **Environment mapping**
- Catmull-Clark Subdivision Surfaces

Question No: 163

A polygon is usually defined by a sequence of vertices and -----

- **Edges (Page 149)**
- Vertices
- Points Ending lines

Question No: 164

AL-JUNAID TECH INSTITUTE

Gouraud shading is the current de jure shading standard in accelerated ----- hardware.

- 2Dimensional
- 3Dimensional (Page 248)
- Multidimensional
- None

Question No: 165

_____ is based on characteristic size or scale

- Fractal Geometry (Fractal shapes are self similar and independent of size or scaling)
- Traditional Geometry
- Euclidean Geometry (Euclidean shapes normally have a few characteristic sizes or lengthscales) (Page 361)
- None of Above

Question No: 166

Arrange the scene to be photographed into the desired composition

- projection transformation
- viewport transformation
- modeling transformation (Page 375)
- viewing transformation

Question No: 167

Which language API defines graphics operations independent of the operating system or computer hardware? Additional hardware specific libraries are used to provide an interface between API and the hardware and between the user and the platform specific windowing system.

- DirectX
- Graphic Windowing Toolkit
- CGI

AL-JUNAID TECH INSTITUTE

- OpenGL (Page 305)

Question No: 168

When transforming a random Axis-Aligned Bounding Box defined by the points (nearx, neary, nearz) and (farx, fary, farz) to the standard orthographic viewing box, which affine transforms are used?

- shear and translation
- rotation and scale
- scale and shear
- Translation and scale

Question No: 169

Shadow mask methods can display a _____ range of colors.

- Small
- Wide (Page 28)
- Random
- Crazy

Question No: 170

Each number that makes up a matrix is called an _____ of the matrix.

- Element (Page 109)
- Variable
- Value
- Component

0304-1659294

www.vulmshelp.com



CS602- Computer Graphics
Solved MCQS
From Finalterm Papers

Feb 27,2014

MC100401285

www.ranapk.com

PSMD01

FINALTERM EXAMINATION
Spring 2010
CS602- Computer Graphics

Question No: 1 (Marks: 1) - Please choose one

Which of the following is NOT a modern application for Computer Graphics-----

- ▶ **Stop-motion animation (Page 6)**
- ▶ Computer Aided Geometric Design
- ▶ Video Games
- ▶ Scientific Visualization

Question No: 2 (Marks: 1) - Please choose one

Both Boundary Filling and Flood filling algorithms are non-recursive techniques,

- ▶ **False [click here 4 detail](#)**
- ▶ True

Question No: 3 (Marks: 1) - Please choose one

TV series are made as simply as possible from the animation point of view. This approach is generally known as -----.

- ▶ Full animation
- ▶ **Limited animation (Page 428)**
- ▶ Low animation
- ▶ High resolution

Question No: 4 (Marks: 1) - Please choose one

An eight frame run cycle that ----- frame/frames to each step gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need dry brush or speed lines to make the movement flow.

- ▶ Two
- ▶ One
- ▶ Three
- ▶ **Four (Page 437)**

Question No: 5 (Marks: 1) - Please choose one

----- reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface.

- ▶ Forward scattering
- ▶ Diffuse Lambertian
- ▶ Backscattering
- ▶ **Retro (Page 293)**

Question No: 6 (Marks: 1) - Please choose one

What makes this really challenging to model is that the index of refraction for most materials is a function of the----- of the light. This means that not only is there a shift in the angle of refraction, but that the shift is different for differing -----of light.

- ▶ Reflecting angle, Reflecting angle
- ▶ Refracting angle, Refracting angle
- ▶ Frequency, Frequency
- ▶ **Wavelength, Wavelength (Page 229)**

Question No: 7 (Marks: 1) - Please choose one

The reflected light wave turns out to be a -----case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Abnormal
- ▶ Complex
- ▶ **Simple (Page 296)**
- ▶ Unknown

Question No: 8 (Marks: 1) - Please choose one

Tessellation can be adaptive to the _____ degree of curvature of a surface.

- ▶ **Local (Page 170)**
- ▶ Static
- ▶ Global
- ▶ Variable

Question No: 9 (Marks: 1) - Please choose one

_____ sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped.

- ▶ glutMainLoop
- ▶ glutIdleFunc
- ▶ **glutReshapeFunc (Page 312)**
- ▶ glutDisplayFunc

Question No: 10 (Marks: 1) - Please choose one

Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -----
-. Floating-point values are mapped directly.

- ▶ -1.0
- ▶ 0.0
- ▶ 2.0
- ▶ **1.0 (Page 320)**

Question No: 11 (Marks: 1) - Please choose one

NURBS stands for-----.

- ▶ Non Universal Rational Binary Spline
- ▶ Non Uniform Rational Binary Splines
- ▶ **Non Uniform Rational Beta Splines (Page 325)**
- ▶ Non Universal Rational Beta Splines

Question No: 12 (Marks: 1) - Please choose one

An orthogonal set of vectors-----

- ▶ Must be a set of linearly independent vectors
- ▶ Must be a set of linearly dependent vectors
- ▶ **Must be made up of the basis vectors (e1, e2, and e3) [click here 4 details](#)**
- ▶ Can be made up of any set of vectors

Question No: 13 (Marks: 1) - Please choose one

Bezier curve is numerically the ----- of all the polynomial-based curves used in these applications.

- ▶ None of the given
- ▶ **Most stable (Page 338)**
- ▶ Less stable
- ▶ Most unstable

Question No: 14 (Marks: 1) - Please choose one

Bezier curve is the ideal standard for representing the ----- piecewise polynomial curves.

- ▶ None of the given
- ▶ Non complex
- ▶ Most complex
- ▶ **More complex (Page 338)**

Question No: 15 (Marks: 1) - Please choose one

Keep polygon orientations consistent to make sure that when viewed from the outside, all the polygons on the surface are oriented in the [same](#) direction.

- ▶ None of the given
- ▶ Neither
- ▶ Different
- ▶ **Same (page 345)**

Question No: 16 (Marks: 1) - Please choose one

The ----- is most simple example that exhibits the property self similarity.

- ▶ Mosse
- ▶ **Fern (Page 355)**
- ▶ None of the given
- ▶ Thohar

Question No: 17 (Marks: 1) - Please choose one

A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize ----- space.

- ▶ Multi-dimensional
- ▶ One-dimensional
- ▶ Two-dimensional
- ▶ **Three-dimensional (Page 371)**

Question No: 18 (Marks: 1) - Please choose one

Which of the following properties of rational Bezier curves fails if the weight assigned to a control point is negative?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ **Convex-Hull [Click here 4 detail](#)**

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Moderen Studies
(IEMS) Samundari**

Question No: 19 (Marks: 1) - Please choose one

In the Phong reflection model, there are 3 constants (a, b, c) which are used to describe the qualities of which of the following phenomena?

- ▶ **The attenuation of a point light source with distance**
- ▶ The size (in each dimension) which the light is assumed to have
- ▶ The amount to perturb reflection vectors as they are mirrored across the normal
- ▶ The material reaction to ambient, diffuse and specular light (respectively)

Question No: 20 (Marks: 1) - Please choose one

The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

- ▶ **Specular**
- ▶ Diffuse
- ▶ Ambient

Question No: 21 (Marks: 1) - Please choose one

When you hit a surface in ray tracing, generally shadow rays are tested against all objects in a scene. If these rays come back saying they hit an object in the scene, which of the following do you do?

- ▶ add all components (i.e. ambient, diffuse and specular) from that light source to the object.
- ▶ add all EXCEPT the ambient light from that light source to the object (i.e. diffuse and specular)
- ▶ **add only the ambient light from that light source to the object**
- ▶ add none of the light from that light source to the object

Question No: 22 (Marks: 1) - Please choose one

The ColorSpace tool is a handy tool that we can use to interactively add two colours together to see the effects of the various strategies for handling oversaturated colours.

- ▶ False
- ▶ **True (Page 235)**

Question No: 23 (Marks: 1) - Please choose one

A polygon is usually defined by a sequence of ----- and Edges.

- ▶ Ending lines
- ▶ Points
- ▶ **Vertices (Page 248)**
- ▶ Edges

Question No: 24 (Marks: 1) - Please choose one

Which of the following properties of Bezier curves guarantees that a line passes through the control polygon as many times or more times than the line passes through the Bezier curve itself?

- ▶ End-point interpolation
- ▶ **Variation Diminishing**
- ▶ Symmetry
- ▶ Convex-Hull

Question No: 25 (Marks: 1) - Please choose one

Parity is a concept used to determine which _____ lie within a polygon. (Choose best suitable answer)

- ▶ Edge
- ▶ Vertices
- ▶ **Pixel (Page 80)**
- ▶ None of the given

Question No: 26 (Marks: 1) - Please choose one

The actual filling process in boundary filling algorithm begins when a point _____ of the figure is selected.

- ▶ Outside the boundary
- ▶ **Inside the boundary (Page 102)**
- ▶ At boundary
- ▶ None of the given

Question No: 27 (Marks: 1) - Please choose one

Weiler-Atherton Polygon Clipping technique modifies the vertex-processing procedures for window boundaries so that _____ polygons are displayed correctly.

- ▶ Convex
- ▶ **Concave (Page 155)**
- ▶ Complex
- ▶ None of the given

Question No: 28 (Marks: 1) - Please choose one

If a line connecting any two points within a polygon does not intersect any edge, then it will be a _____ polygon.

- ▶ **Convex (Page 79)**
- ▶ Concave
- ▶ Complex
- ▶ None of the given

Question No: 29 (Marks: 1) - Please choose one

_____ can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the view plane which constitutes the display surface.

- ▶ Mapping plane
- ▶ Three Coordinate Planes
- ▶ View plane
- ▶ **Projection (Page 193)**

Question No: 30 (Marks: 1) - Please choose one

The reflected light wave turns out to be a / an _____ case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Unknown
- ▶ **Simple (Page 296)**
- ▶ Complex
- ▶ Abnormal

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Modern Studies
(IEMS) Samundari**

FINAL TERM EXAMINATION
Spring 2010
CS602- Computer Graphics

Question No: 1 (Marks: 1) - Please choose one

Computer graphics and computer vision are _____ disciplines.

Related (Page 24)

Interrelated

Same

Different

Question No: 2 (Marks: 1) - Please choose one

OpenGL has become the industry's most widely used and supported _____ graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms.

2-Dimensional

3-Dimensional

2-Dimensional and 3-Dimensional (Page 301)

Question No: 3 (Marks: 1) - Please choose one

----- sets the global idle callback to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

glutIdle function (Page 313)

glutKeyboardFunc

glutReshapeFunc

glutDisplayFunc

Question No: 4 (Marks: 1) - Please choose one

$X^2/a^2 - y^2/b^2 = 1$ is an equation of

Hyperbola (Page 70)

Parabola ($4px=y^2$)

None of given

Ellipse ($X^2/a^2 + y^2/b^2 = 1$)

Question No: 5 (Marks: 1) - Please choose one

To modify the object shape, shearing transformation cannot be used.

False (Page 192)

True

Question No: 6 (Marks: 1) - Please choose one

Rotation is performed around a fixed point called _____.

Pivot point rotation (Page 119)

Question No: 7 (Marks: 1) - Please choose one

Computer graphics is very helpful in producing graphical representations for scientific visualization and analysis

True (Page 9)

False

Question No: 8 (Marks: 1) - Please choose one

Save a line with both endpoints inside all clipping boundaries.

Trivial Reject

Trivial Accept (Page 142)

None of given

Question No: 9 (Marks: 1) - Please choose one

Dark lights are nothing more than lights in which one or more of the color values are _____.

Unknown

Negative (Page 235)

Positive

Zero

Question No: 10 (Marks: 1) - Please choose one

A series of _____ computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection.

Three (Page 371)

Two

Four

Ten

Question No: 11 (Marks: 1) - Please choose one

At a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our _____ cells.

Eye

Retina

Cone (Page 398)

Question No: 12 (Marks: 1) - Please choose one

This projection technique has the direction of projection perpendicular to the viewing plane, but the viewing direction is NOT perpendicular to one of the principle faces.

Orthographic Parallel Projection

Axometric Parallel Projection (Page 194)

Oblique Parallel Projection

Question No: 13 (Marks: 1) - Please choose one

In the Phong Reflection model, _____ light is the same everywhere.

Ambient

Diffuse

Specular

Emissive

Question No: 14 (Marks: 1) - Please choose one

A plane is two dimensional since in order to uniquely define any point on its surface we require _____ numbers.

Two (Page 356)

Three

Four

Five

Question No: 15 (Marks: 1) - Please choose one

In perspective projection, for your view to come out correctly, you will also want the _____ to pass through the middle of the screen.

X axis

Y axis

Z axis (Page 200)

None

Question No: 16 (Marks: 1) - Please choose one

Neither floating-point nor signed integer values are clamped to the range _____ before updating the current color.

0 , -1.0

-1 , 1

1 , -1

0, 1 (Page 321)

Question No: 17 (Marks: 1) - Please choose one

An object's _____ determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source.

Unit

Normal (Page 400)

None of given

Question No: 18 (Marks: 1) - Please choose one

Which was the oldest shading model?

Flat Shading (Page 245)

Phong Shading

Gouraud Shading

Question No: 19 (Marks: 1) - Please choose one

Which of the following affine transforms does NOT affect vectors?

Scale

Rotation

Shear

Translation

Final Term MCQS and Quizzes

Question # 1 of 10 (Total Marks: 1) Select correct option:

We want our scene to look more realistic, we should use _____ lights.

Ambient (Page 282)

Point Parallel

Spot

None of the given

Question # 2 of 10 (Total Marks: 1) Select correct option:

This is a simple example of line clipping: the display window is the canvas and also the default -----
-, thus all line segments inside the canvas are drawn.

Clipping Rectangle (Page 141)

Clipping Circle

Clipping Polygon

Clipping Angle

Question # 3 of 10 (Total Marks: 1) Select correct option:

One problem with Gouraud shading is that the ----- intensities can never be greater than the intensities at the edges.

Triangles (Page 246)

Squares

Rectangles

Polygons

Question # 4 of 10 (Total Marks: 1) Select correct option:

There is more penetration of light in case of _____ surfaces.

Conductor (like metals)

Nonconductor (like dielectrics) (Page 235)

Both conductor and nonconductor

None of the given

Question # 5 of 10 (Total Marks: 1) Select correct option:

_____ lights should be avoided because they are not for real time environment.

Point

Parallel

Spot (Page 244)

None of the given

Question # 6 of 10 (Total Marks: 1) Select correct option:

The physical range of colors a device can display is called _____ of the device.

Sharpness

Gamut (Page 229)

Colouring

Colouring with Sharpness

Question # 7 of 10 (Total Marks: 1) Select correct option:

_____ is simply the calculation of color reflected by the surface.

Shading (Page 240)

Clamping

Scaling

None of the given

Question # 8 of 10 (Total Marks: 1) Select correct option:

When obtaining normals for a triangle, which of the following mathematical constructs is NOT used?

Vector normalization

Vector cross products

Vector dot products

Point-Point subtraction

Question # 9 of 10 (Total Marks: 1) Select correct option:

Loosely, the alpha component of the RGBA quad represents the _____ of a surface.

Opacity (Page 227)

Light

Darkness

Shine

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Modern Studies
(IEMS) Samundari**

Question # 10 of 10 (Total Marks: 1) Select correct option:

An algorithm that clips a polygon must deal with many ----- cases. The case is particularly noteworthy in that the concave polygon is clipped into ----- isolate polygons.

Similar, three

Different, two (Page 146)

Different, three

Similar, two

Question # 1 of 10 (Total Marks: 1) Select correct option:

_____ lighting is not dependent on any source.

Ambient [Click here for detail](#)

Diffuse

Specular

Emissive

Question # 2 of 10 (Total Marks: 1) Select correct option:

In order to get a more realistic representation of lighting, we'll need to understand how light passes through a medium and how hitting the boundary layer at the ----- of two media can affect light's properties.

Intersection (Page 296)

Union

Endpoints

Edges

Question # 3 of 10 (Total Marks: 1) Select correct option:

Lambertian shading was used mostly back when computers weren't fast enough to do _____ in real time.

Phong shading

Processing

Shading

Gouraud shading (Page 245)

Question # 4 of 10 (Total Marks: 1) Select correct option:

In Perspective Projection the point of View (POV) must lie on the _____.

All axis

Z axis (Page 200)

X axis

Y axis

Question # 5 of 10 (Total Marks: 1) Select correct option:

If we want any object to glow, we should use _____ lights.

Ambient

Diffuse

Specular

Emissive (Page 240)

Question # 6 of 10 (Total Marks: 1) Select correct option:

There are not many different ways of representing the intensity of a particular color element.

True

False (Page 276)

Question # 7 of 10 (Total Marks: 1) Select correct option:

In Perspective Projection the screen plane must be parallel to the_____.

Y-Z plane

X-Y plane (Page 200)

Z-Y plane

X-Z plane

Question # 8 of 10 (Total Marks: 1) Select correct option:

_____ light is reflected in all directions from surface.

Ambient

Diffuse (page 239)

Specular

Emissive

Question # 9 of 10 (Total Marks: 1) Select correct option:

A space curve can be confined to a plane.

Yes

No (Page 331)

Question # 10 of 10 (Total Marks: 1) Select correct option:

To convert the information in the A matrix into that required for the P matrix, we do some simple matrix algebra, First we have $UA=UNP$ then Simply $A = \text{-----}$

UP

NP (Page 333)

UN

None

Question # 1 of 10 (Total Marks: 1) Select correct option:

Perspective projection is specified with the function glFrustum().

Yes (Page 376)

No

Question # 2 of 10 (Total Marks: 1) Select correct option:

Choose a camera lens or adjust the zoom

projection transformation (Page 372)

viewport transformation

modeling transformation'

viewing transformation

Question # 3 of 10 (Total Marks: 1) Select correct option:

Using a lighting model based upon the Blinn Phong model means that we'll always get a uniform specular highlight based upon the color of the ----- light and material, which means that all reflections based on this model, will be reminiscent of plastic.

Union

Refracting

Intersection

Reflecting (Page 296)

Question # 4 of 10 (Total Marks: 1) Select correct option:

Refractive index is a function of temperature, mostly due to density changes in materials with changes in temperature.

True (Page 300)

False

Question # 5 of 10 (Total Marks: 1) Select correct option:

Length L depends on the angle alpha and the z coordinate of the point to be projected and L can be represented by -----.

$z * 1/ \tan (\alpha)$ (Page 198)

$z * L^2$

$z * 1/ \tan (\beta)$

$z * 1/ \tan (\gamma)$

Question # 6 of 10 (Total Marks: 1) Select correct option:

The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the _____ light.

Ambient

Ambient, diffuse, and specular (Page 281)

Specular

Diffuse, and specular

Question # 7 of 10 (Total Marks: 1) Select correct option:

Another way to define a space curve by using intermediate points and the tangents at each end for making the curve

Yes

No (Page 334)

Question # 8 of 10 (Total Marks: 1) Select correct option:

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, ----- graphics standard.

Tertiary

Binary

Single platform

Multiplatform (Page 301)

Question # 9 of 10 (Total Marks: 1) Select correct option:

glutReshapeWindow requests a change in the size of the current window. The width and height parameters are size extents in pixels. The width and height must be ----- values.

Neutral

Negative

Positive (Page 311)

None of the given

Question # 10 of 10 (Total Marks: 1) Select correct option:

A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are -----.

Binary polynomials

Mono polynomials

Quadratic polynomials

Cubic polynomials (Page 331)

Question # 1 of 10 (Total Marks: 1) Select correct option:

Refractive index is a function of temperature, mostly due to changes in ----- of materials with changes in temperature. A simple correction can be applied in most circumstances to allow us to use a value given at one temperature at another.

Density (Page 300)

pressure
nature
volume

Question # 2 of 10 (Total Marks: 1) Select correct option:

If we assign a different value to the parametric variable for the intermediate point, then we obtain different values for the coefficients. This, in turn, means that a different curve is produced, although it passes through the ----- three points.

isolate
different

same (Page 328)

none

Question # 3 of 10 (Total Marks: 1) Select correct option:

The attenuation formula is $f = \frac{1}{C + Ld + Qd^2}$, where C, L and Q are the constant, linear and quadratic attenuation factors and d is the distance between the vertex being lit and the light source.

$\frac{1}{C + Ld + Qd^2}$ [Click here for detail](#)

$\frac{1}{C + Ld + Qd}$
 $\frac{1}{C + L + d + Qd^2}$
 $\frac{1}{Cd + Ld + Qd^2}$

Question # 4 of 10 (Total Marks: 1) Select correct option:

Bezier curve is tangent to the lines connecting _____.

First two points
Last two points

First two points and last two point (Page 340)

None of the given

Question # 5 of 10 (Total Marks: 1) Select correct option:

End points and an intermediate point on the curve, then we now ----- quantities that we can express in terms of these coefficients (3 points x 3 coordinates each), and we can use these three points to define a unique curve.

Six
Three
Two

Nine (Page 326)

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Modern Studies
(IEMS) Samundari**

Question # 6 of 10 (Total Marks: 1) Select correct option:

Choose a camera lens or adjust the zoom

projection transformation (Page 372)

viewport transformation

modeling transformation

viewing transformation

Question # 7 of 10 (Total Marks: 1) Select correct option:

_____ OpenGL function is used for aiming and positioning the camera towards the object

glLoadIdentity() (Page 375)

gluLookAt()

glFrustum()

None of Above

Question # 8 of 10 (Total Marks: 1) Select correct option:

A parametric curve is one whose defining equations are given in terms of a -----, common, independent variable called the parametric variable.

Triple

Double

Single (Page 325)

None of the given

Question # 9 of 10 (Total Marks: 1) Select correct option:

The reflection coefficients are in the ----- range and are specified as part of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient.

[0, 10]

[0, 1] (Page 281)

[0, 5]

[0, 2]

Question # 10 of 10 (Total Marks: 1) Select correct option:

To ensure a smooth transition from one section of a piecewise _____ to the next, we can impose various continuity conditions at the connection points

non parametric curve

parametric curve

polygon vector (not confirm) (Page 245)

None of the these

Question # 1 of 10 (Total Marks: 1) Select correct option:

The curve is always contained within the _____ of the control points

Tangents

Convex Hull (Page 340)

Subdivision

None of Above

Question # 2 of 10 (Total Marks: 1) Select correct option:

Projection can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the -----, which constitutes the display surface. The mapping is determined by a projection line called the projector that passes through P and intersects the -----.

Two Coordinate Planes

View plane or projection plan (Page 193)

Three Coordinate Planes

Mapping plane

Question # 3 of 10 (Total Marks: 1) Select correct option:

Determine how large we want the final photograph to be - for example, we might want it enlarged

projection transformation

viewport transformation (Page 372)

modeling transformation

viewing transformation

Question # 4 of 10 (Total Marks: 1) Select correct option:

Ambient light is the light that comes from -----directions, thus all surfaces are illuminated equally regardless of orientation. However, this is a big hack in traditional lighting calculations since "real" ambient light really comes from the light reflected from the "environment."

All (Page 281)

Opposite

Same

Four different

Question # 5 of 10 (Total Marks: 1) Select correct option:

Silhouette edges occur when dot product of surface normal vector and the view vector is _____.

Zero (Page 345)

One

Both zero and one

Question # 6 of 10 (Total Marks: 1) Select correct option:

If the current matrix (according to glMatrixMode) is multiplied by the translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with -----.

M-T

M+T

M/T

M*T (Page 317)

Question # 7 of 10 (Total Marks: 1) Select correct option:

Arrange the scene to be photographed into the desired composition

projection transformation

viewport transformation

modeling transformation (Page 317)

viewing transformation

Question # 8 of 10 (Total Marks: 1) Select correct option:

In the forms of texture mapping, Image to world space and world space to image, each suffers from different problems related to minification and magnification. Which of the two shows the following problem: When the texture is larger than the screen space it maps to, many texture units (texels) are never sampled?

Image to world space

World space to image

X-axis

Y-axis

Question # 9 of 10 (Total Marks: 1) Select correct option:

Imagine a curve in three-dimensional space, each point on the curve has a unique set of coordinates: a specific x value, y value, and z value. Each coordinate is controlled by a ----- parametric equation.

Opposite

Similar

Separate (Page 325)

Question # 10 of 10 (Total Marks: 1) Select correct option:

We allow the parametric variable to take on values only in the interval -----.

$-1 \leq u \leq 0$

$0 \leq u \leq 2$

$0 \leq u \leq 1$ (Page 326)

$-1 \leq u \leq 1$

Question # 1 of 10 (Total Marks: 1) Select correct option:

Bezier curve can represent the more complex piecewise _____ curve.

Polynomial (Page 338)

Exponential

Cubic

None of above

Question # 2 of 10 (Total Marks: 1) Select correct option:

A fractal is generally a property called _____.

Fractal Dimension

Self-similarity (Page 355)

Koch Curve

None of above

Question # 3 of 10 (Total Marks: 1) Select correct option:

Normalized cross product of two vectors on that surface provides normal vector

Yes (Page 347)

No

Question # 4 of 10 (Total Marks: 1) Select correct option:

Every point on a curve has a straight line associated with it called the _____

State line

tangent line (Page 334)

curved line

None of the given

Question # 5 of 10 (Total Marks: 1) Select correct option:

The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at ----- . This window identifier can be used when calling glutSetWindow.

Three

Two

One (Page 308)

Zero

Question # 6 of 10 (Total Marks: 1) Select correct option:

Curve and surface equations can be expressed in either a parametric or a non parametric form.

True [click here for detail](#)

False

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Moderen Studies
(IEMS) Samundari**

Question # 7 of 10 (Total Marks: 1) Select correct option:

Bernstein polynomial functions are the basic functions of _____ curves.

NURBS

Bezier (Page 342)

Both NURBS and Bazier

None of the given

Question # 8 of 10 (Total Marks: 1) Select correct option:

Geometric patterns that is repeated at ever smaller scales to produce irregular shapes and surfaces are called

Geometric patterns

Fractals (Page 352)

Animated components

Segments

Question # 9 of 10 (Total Marks: 1) Select correct option:

The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Each piece of the Phong model uses different vectors and constants. Which portion does NOT include taking a dot product?

Ambient

Diffuse

Specular

Question # 10 of 10 (Total Marks: 1) Select correct option:

This projection technique does NOT have the direction of projection perpendicular to the viewing plane.

a) Orthographic Parallel Projection

b) Axonometric Parallel Projection

c) Oblique Parallel Projection (Page 197)

Question # 1 of 10 (Total Marks: 1) Select correct option:

In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with a glRotatef function call?

a) GL_MODELVIEW

b) GL_PROJECTION

Question # 2 of 10 (Total Marks: 1) Select correct option:

In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with glFrustum?

- a) GL_MODELVIEW
- b) GL_PROJECTION**

Question # 3 of 10 (Total Marks: 1) Select correct option:

Which of the following is the order that geometry operations are performed in OpenGL (where we read the order from left to right)?

- a) GL_PROJECTION GL_MODELVIEW Perspective division
- b) GL_MODELVIEW GL_PROJECTION Perspective division**
- c) Perspective division GL_PROJECTION GL_MODELVIEW
- d) GL_MODELVIEW Perspective division GL_PROJECTION
- e) GL_PROJECTION Perspective division GL_MODELVIEW

Question # 4 of 10 (Total Marks: 1) Select correct option:

Which of the following is NOT true about quaternions?

- a) They are made up of 4 numbers
- b) They should always be normalized to length 1
- c) They can be used to represent all affine transforms**
- d) They can be used to define the rotation of an object

Question # 5 of 10 (Total Marks: 1) Select correct option:

In class, we discussed three forms of shading for “Utah” graphics. Which was the first to use per vertex normals?

- a) Flat Shading (Page 245)**
- b) Phong Shading
- c) Gouraud Shading

Question # 6 of 10 (Total Marks: 1) Select correct option:

Given any implicit equation, which of the following is true for all (x, y, z) that make the equation exactly zero?

- a) All those points are inside the object defined by the implicit equation
- b) All those points are on the surface of the object defined by the implicit equation** [Click here 4 detail](#)
- c) All those points are outside the object defined by the implicit equation
- d) You can't know anything without knowing what the implicit equation is

Question # 7 of 10 (Total Marks: 1) Select correct option:

When solving ray-sphere intersections using the implicit equation for a sphere, you must solve the quadratic equation. Which of the following do you know if the B^2-4AC (i.e. the part under the square root) is negative?

- a) The ray intersects the sphere at a negative t... discard this result
- b) The ray intersects the sphere at a positive t... continue to the solution
- c) The ray does not intersect the sphere... discard this result**
- d) The ray begins inside the sphere... this is a special case

Question # 8 of 10 (Total Marks: 1) Select correct option:

_____ sets the global idle call back to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

glutIdleFunc (Page 313)

- glutMainLoop
- glutDisplayFunc
- glutReshapeFunc

Question # 9 of 10 (Total Marks: 1) Select correct option:

A tangent vector certainly defines the slope at one end of the curve, but a vector has characteristics of.....

- direction
- magnitude

both direction and magnitude (Page 336)

- None of the given

Question # 10 of 10 (Total Marks: 1) Select correct option:

The degree of a Bezier curve is equal to $n-1$, where n is the number of control points

Yes (Page 339)

- No

Question # 1 of 10 (Total Marks: 1) Select correct option:

Bit mask to select a window with multisampling support. If multisampling is not available, a ----- window will automatically be chosen.

Non-multisampling (Page 310)

- Multisampling
- Mono-multisampling
- Di-multisampling

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Modern Studies
(IEMS) Samundari**

Question # 2 of 10 (Total Marks: 1) Select correct option:

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers ----- information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

Encapsulate (Page 302)

- Shows
- Hibernates
- None of the given

Question # 3 of 10 (Total Marks: 1) Select correct option:

With similar expressions for $y(u)$ and $z(u)$. Again the a, b, c and d terms are constant coefficients. As we did with Equation for a plane curve, we combine the $x(u), y(u)$, and $z(u)$ expressions into a single vector equation $P(u) =$ -----.

- $Au^2+bu+cu+d$
- $Au^4+bu^3+cu^2+d$
- Au^3+bu^2+cu+d

Au^3+bu^2+cu+d (Page 331)

Question # 4 of 10 (Total Marks: 1) Select correct option:

Each number that makes up a matrix is called an _____ of the matrix.

Element (Page 106)

- Variable
- Value
- Component

Question # 5 of 10 (Total Marks: 1) Select correct option:

The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to `gluPerspective` with a call to -----.

`glRotated`
`gluPerspective` (Page 318)

- `glTranslated`
- `glLoadIdentity`

Question # 6 of 10 (Total Marks: 1) Select correct option:

Which one of the following step is not involved to write pixel using video BIOS services.

- Setting desired video mode
- Using BIOS service to set color of a screen pixel
- Calling BIOS interrupt to execute the process of writing pixel.
- Using OpenGL service to set color of a screen pixel (Page 45)**

Question # 7 of 10 (Total Marks: 1) Select correct option:

Shadow mask methods can display a _____ range of colors.

Small

Wide (Page 28)

Random

Crazy

Question # 8 of 10 (Total Marks: 1) Select correct option:

Using Cohen-Sutherland line clipping, it is impossible for a vertex to be labeled 1111.

True

False

Question # 9 of 10 (Total Marks: 1) Select correct option:

Intensity of the electron beam is controlled by setting _____ levels on the control grid, a metal cylinder that fits over the cathode.

Amplitude

Current

Voltage (Page 26)

Electron

Question # 10 of 10 (Total Marks: 1) Select correct option:

The circle and ellipse are symmetric across 8 octants.

True

False (Page 60)

Question # 1 of 10 (Total Marks: 1) Select correct option:

UV light is used in Plasma Panel displays to excite phosphor.

True [Click here for detail](#)

False

Question # 2 of 10 (Total Marks: 1) Select correct option:

Which of the following is not true about matrices?

$A + B = B + A$

$a(A + B) = aA + aB$

$(AT)T = AT$

$A + (B + C) = (A + B) + C$

Question # 3 of 10 (Total Marks: 1) Select correct option:

According to Odd Parity Rule, a point is inside the polygon, if:

- Line from an outside point to this point does not cross the edges odd number of times
- Line from any point to this point crosses the edges odd number of times
- Line from an outside point to this point crosses the edges odd number of times (Page 80)**
- Line from this point to any point outside the polygon intersects any edge

Question # 4 of 10 (Total Marks: 1) Select correct option:

As opposed to direct memory access method, BIOS routines provide an easier and faster method of drawing pixels on screen.

- True
- False (Page 47)**

Question # 5 of 10 (Total Marks: 1) Select correct option:

When a point P(x,y) is rotated by θ the coordinates of transformed point P' are given as:

$x' = x \cos(\theta) - y \sin(\theta)$, $y' = x \sin(\theta) + y \cos(\theta)$ (Page 181)

$x' = y \cos(\theta) - x \sin(\theta)$, $y' = y \sin(\theta) + x \cos(\theta)$

$x' = x \cos(\theta) + y \sin(\theta)$, $y' = x \sin(\theta) - y \cos(\theta)$

$x' = y \cos(\theta) + x \sin(\theta)$, $y' = y \sin(\theta) - x \cos(\theta)$

Question # 6 of 10 (Total Marks: 1) Select correct option:

The amount of time it takes to illuminate a specific location on phosphor coated screen is called Persistence.

- True
- False (Page 27)**

Question # 7 of 10 (Total Marks: 1) Select correct option:

Incremental line drawing algorithm makes use of the equation of straight line.

- True
- False (Not sure)**

Question # 8 of 10 (Total Marks: 1) Select correct option:

In matrix multiplication:

- The two matrices must be square
- The number of rows of 1st matrix must be the same as the number of columns of the second.**
- The two matrices must either be row matrices or column matrices
- The number of columns of 1st matrix must be the same as the number of rows of the second.

Question # 9 of 10 (Total Marks: 1) Select correct option:

In Horizontal retrace, after completion of all the pixels in a scan line, the refreshing continues from the 1st pixel of the next scan line.

True

False (Page 28)

Question # 10 of 10 (Total Marks: 1) Select correct option:

Parity Rule is used to determine whether a pixel is inside a polygon or not.

True (Page 80)

False

Question # 1 of 10 (Total Marks: 1) Select correct option:

When dot product of two vectors equals zero, this implies that the two vectors are:
parallel to each other

orthogonal (perpendicular) to each other. (Page 177)

intersect each other

equal to each other

Question # 2 of 10 (Total Marks: 1) Select correct option:

In Pixmap exactly one bit is used to hold color value of each pixel.

True

False (Page 28)

Question # 3 of 10 (Total Marks: 1) Select correct option:

The Boundary Fill and Flood Fill algorithms:

Must use 4-connected approach

Must use 8-connected approach

May use 4-connected or 8-connected approach [click here for details](#)

Must not use recursive approach

Question #4 of 10 (Total Marks: 1) Select correct option:

To show 256 colors, the no of bits required for each pixel are

a. 8 (Page 39)

b. 16

c. 32

d. 64

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Modern Studies
(IEMS) Samundari**

Question # 5 of 10 (Total Marks: 1) Select correct option:

25 * 80 resolution with 16 colors supports

a. Text mode (Page 43)

- b. Graphics mode
- c. Both
- d. None

Question # 6 of 10 (Total Marks: 1) Select correct option:

Two matrices are said to be equal, if they have

- a. same order
- b. same corresponding elements
- c. Same order and same corresponding elements.**
- d. Different elements.

Question # 7 of 10 (Total Marks: 1) Select correct option:

Two points are said to be collinear, if they lie on the

a. same line (page 53)

- b. different but parallel lines
- c. either on the same plane or two parallel planes
- d. different plane

Question # 8 of 10 (Total Marks: 1) Select correct option:

A Polygon is convex, if the line connecting:

- a. Any two points outside the polygon intersects its boundary
- b. Any two points inside the polygon don't intersect any edges of the polygon. (Page 78)**
- c. A point inside the boundary with any point outside, does not intersect the polygon boundary
- d. Any two vertices, intersects some edge of polygon.

Question # 9 of 10 (Total Marks: 1) Select correct option:

Polygon is complex, if the line connecting:

- a. Any two points outside the polygon intersects its boundary.
- b. Any two points inside the polygon intersects its boundary.
- c. A point inside the boundary with any point outside does not intersect the polygon boundary.
- d. Any two vertices, intersects some edge of polygon. (Page 79)**

Muhammad Moaaz Siddiq – MCS(4th)

Moaaz.pk@gmail.com

**Campus:- Institute of E-Learning & Moderen Studies
(IEMS) Samundari**

Question # 10 of 10 (Total Marks: 1) Select correct option:

The equation of hyperbola centered at origin (if the transverse axis is along x -axis) can be given as:

- a. $x^2/b^2 + y^2/a^2 - 1 = 0$
- b. $x^2/b^2 + y^2/a^2 + 1 = 0$
- c. $x^2/a^2 - y^2/b^2 - 1 = 0$ [Click here for detail](#)**
- d. $x^2/b^2 - y^2/a^2 - 1 = 0$

Question # 1 of 10 (Total Marks: 1) Select correct option:

Which one is not valid out code to perform trivial accept / reject test in line clipping:

- a. 1101
- b. 1001 (Page 143)**
- c. 0101
- d. 0110

Question # 2 of 10 (Total Marks: 1) Select correct option:

Which one of the following is not the graphics library is use:

- a. FastGL
- b. OpenGL
- c. DirectX
- d. EasyGL (Page 42)**

Question # 3 of 10 (Total Marks: 1) Select correct option:

In video text memory, _____ are used to display a character.

2 bytes (Page 43)

- 4 bytes
- 8 bytes
- 16 bytes

Question # 4 of 10 (Total Marks: 1) Select correct option:

In _____ algorithm(s), old color must be read before it is invoked.

- Scan line filling
- Flood filling (Page 104)**
- Both scan line and flood filling
- Scan filling

Question # 5 of 10 (Total Marks: 1) Select correct option:

According to the architecture of raster graphics system, display processor memory will act as_____.

Video controller (Page 36)

System memory

Frame buffer

Video controller and System memory

Question # 6 of 10 (Total Marks: 1) Select correct option:

Global coordinate systems can be defined with respect to local coordinate system

True

False (Page 255)

Question # 7 of 10 (Total Marks: 1) Select correct option:

Cross product of two vectors result in a _____.

Magnitude

Vector (Page 347)

Scalar

Value

Question # 8 of 10 (Total Marks: 1) Select correct option:

Shortcoming of Sutherland-Hodgeman Algorithm is concave polygons may be displayed with extraneous lines

True (Page 155)

False

Question # 9 of 10 (Total Marks: 1) Select correct option:

In 2D transformations, rotations applied to a point P can be denoted as_____ (Where? represents theta).

$P' = R(\theta_1 + \theta_2) \cdot P$ (Page 119)

$P' = (R(\theta_1) + R(\theta_2)) \cdot P$

$P' = R(\theta_1 \times \theta_2) \cdot P$

$P' = R(\theta_1) \cdot P$

Question # 10 of 10 (Total Marks: 1) Select correct option:

A straight line can be moved to another location by applying _____ to each of the line endpoints and redrawing the line between the new coordinates.

Rotation

Translation (Page 118)

Reflection

Scaling factor

Question # 1 of 10 (Total Marks: 1) Select correct option:

If the values of scaling factors s_x and s_y are less than 1, then size of object will be _____.

Reduced (Page 121)

- Remain same
- Enlarged
- Shear

Question # 2 of 10 (Total Marks: 1) Select correct option:

Locations can be translated or "transformed" from one coordinate system to the other.

True (Page 163)

- False

Question # 3 of 10 (Total Marks: 1) Select correct option:

If the value of scaling factors s_x and s_y is greater than 1, then size of objects will be _____.

- Reduced

Enlarged (Page 121)

- Remain same
- Shear

Question # 4 of 10 (Total Marks: 1) Select correct option:

A column matrix is also known as _____.

Column vector (Page 107)

- Row vector
- Vector
- Simple Matrix

Question # 5 of 10 (Total Marks: 1) Select correct option:

Dot product of two vectors results in _____ quantity.

Scalar (Page 176)

- Vectors
- Magnitude
- Value

Question # 6 of 10 (Total Marks: 1) Select correct option:

Set up your tripod and pointing the camera at the scene

- projection transformation
- viewport transformation
- modeling transformation

viewing transformation (Page 372)

Question # 7 of 10 (Total Marks: 1) Select correct option:

_____ is based on characteristic size or scale

Fractal Geometry

Traditional Geometry

Euclidean Geometry (Page 359)

None of Above

Question # 8 of 10 (Total Marks: 1) Select correct option:

Bernstein polynomial functions are the basic functions of _____ curves.

NURBS

Bezier (Page 342)

Both NURBS and Bazier

None of the given

Question # 9 of 10 (Total Marks: 1) Select correct option:

Which of the following does NOT figure into the Field of View of a pinhole camera?

a) The direction of projection

b) The distance from the center of projection to the projection plane

c) The size of the projection plane

Question # 10 of 10 (Total Marks: 1) Select correct option:

When transforming a random Axis-Aligned Bounding Box defined by the points (nearx, neary, nearz) and (farx, fary, farz) to the standard orthographic viewing box, which affine transforms are used?

a) shear and translation

b) rotation and scale

c) scale and shear

d) translation and scale

Question # 1 of 10 (Total Marks: 1) Select correct option:

In class, we discussed the purpose of the front and back clipping planes in OpenGL. Which of the following was NOT a purpose for using clipping planes?

a) division by zero

b) objects behind the center of projection mapping onto the projection plane

c) avoiding the problems of infinite viewing volume size

Question # 2 of 10 (Total Marks: 1) Select correct option:

In class, we discussed how the image of the Double Eagle Tanker was obtained for the large poster in the main hall of Sitterson. It required rendering several perspective images using OpenGL. Which of the following was NOT a step required in that process?

- a) handling projection planes non-orthogonal to the viewing direction
- b) cutting a single projection plane into many separate projection planes
- c) rotating the viewing direction to be the same as the $-z$ direction
- d) handling several different centers of projection**

Question # 3 of 10 (Total Marks: 1) Select correct option:

In “Utah” graphics, lights are simplified in order to approximate light/matter interaction with a minimum amount of work. Which of the following is NOT true about the simplifications made in “Utah” graphics lights?

- a) Light intensity and color are folded into one value.
- b) Lights are assumed to have zero size
- c) Spotlights cannot be handled**
- d) Soft shadows cannot be handled

Question # 4 of 10 (Total Marks: 1) Select correct option:

The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which of the following is NOT a vector needed for the Phong reflection model?

- a) Surface Normal
- b) Direction to Viewer
- c) Direction to Material Center**
- d) Direction to Light

Question # 5 of 10 (Total Marks: 1) Select correct option:

In the Phong Reflection model, ambient light is the same everywhere.

- a) true**
- b) false

Question #6 of 10 (Total Marks: 1) Select correct option:

We discussed several global illumination algorithms in class. Which of the following listed can properly handle diffuse-diffuse reflections

- a) Ray Tracing
- b) Radiosity**
- c) Photon Mapping
- d) RenderMan

Question # 7 of 10 (Total Marks: 1) Select correct option:

We discussed several global illumination algorithms in class. Which of the following is generally characterized by shiny spheres and checkerboards?

- a) **Ray Tracing**
- b) Radiosity
- c) Photon Mapping
- d) RenderMan

Question # 8 of 10 (Total Marks: 1) Select correct option:

We discussed several global illumination algorithms in class. Which of the following listed can properly handle caustics?

- a) Ray Tracing
- b) Radiosity
- c) **Photon Mapping**
- d) RenderMan

Question # 9 of 10 (Total Marks: 1) Select correct option:

In radiosity a matrix is formed of size $m \times m$. Which of the following can be known if the (i, j) position in the matrix is zero?

- a) patch i is much larger than patch j
- b) patch i is much smaller than patch j
- c) patch i is visible from patch j
- d) **patch i is not visible from patch j**

Question # 10 of 10 (Total Marks: 1) Select correct option:

When solving for ray-polygon intersections, after intersecting the ray with a plane, the dominant component of the plane normal is found. this is used to

- a) ignore any component other than the dominant when you project to 2D
- b) **ignore the dominant component when you project to 2D**
- c) solve the inside-outside test only for that component

Question # 1 of 10 (Total Marks: 1) Select correct option:

If solving for inside-outside of a triangle using barycentric coordinates, and you have two barycentric coordinates solved ($bc1 = .57, bc2 = .62$) which of the following do you know?

- a) **The point is outside the triangle**
- b) The point is on the boundary of the triangle
- c) The point is inside the triangle
- d) The hit point is on the “back face” of the triangle

Question # 2 of 10 (Total Marks: 1) Select correct option:

The majority of the execution time of a ray tracer is spent in ray-object intersection code.

- a) true
- b) false

Question # 3 of 10 (Total Marks: 1) Select correct option:

Snell's law is used to calculate which one of the following rays?

- a) start rays
- b) shadow rays
- c) reflection rays
- d) transmission rays**

Question # 4 of 10 (Total Marks: 1) Select correct option:

You might as well recurse on reflection and transmission rays until you hit nothing. This always is bound to happen.

- a) true
- b) false**

Question # 5 of 10 (Total Marks: 1) Select correct option:

Which of the following Ray Tracing additions works to remove jaggies alone?

- Mkkg**
- b) jittering

Question # 6 of 10 (Total Marks: 1) Select correct option:

In the Pixar short "Geri's Game", the character Geri was created using which of the following techniques?

- a) Fractals
- b) Bump mapping
- c) Environment mapping
- d) Catmull-Clark Subdivision Surfaces**

Question # 7 of 10 (Total Marks: 1) Select correct option:

In the Pixar short "Geri's Game", the trees in the background were created using which of the following techniques?

- a) Fractals**
- b) Bump mapping
- c) Environment mapping
- d) Catmull-Clark Subdivision Surfaces

Question # 8 of 10 (Total Marks: 1) Select correct option:

The basic functions $f_i(u)$ in Bezier curve must be symmetric with respect to u and $(u-2)$
yes

no (Page 341)

Question # 9 of 10 (Total Marks: 1) Select correct option:

In the Pixar short “Geri’s Game”, Geri’s glasses seemed to bend the light as it passed through.
Which of the following techniques was used?

- a) Fractals
- b) Bump mapping
- c) Environment mapping**
- d) Catmull-Clark Subdivision Surfaces

Question # 10 of 10 (Total Marks: 1) Select correct option:

A polygon is usually defined by a sequence of vertices and -----

Edges (Page 146)

Vertices

Points
Ending lines

Question # 1 of 10 (Total Marks: 1) Select correct option:

Gouraud shading is the current de jure shading standard in accelerated ----- hardware.
2Dimensional

3Dimensional (Page 245)

Multidimensional

None

Question #2 of 10 (Total Marks: 1) Select correct option:

_____ is based on characteristic size or scale

Fractal Geometry (Fractal shapes are self similar and independent of size or scaling)

Traditional Geometry

Euclidean Geometry (Euclidean shapes normally have a few characteristic sizes or length scales) (Page 359)

None of Above

Question # 3 of 10 (Total Marks: 1) Select correct option:

Arrange the scene to be photographed into the desired composition

projection transformation

viewport transformation

modeling transformation (Page 372)

viewing transformation

Question # 4 of 10 (Total Marks: 1) Select correct option:

Which language API defines graphics operations independent of the operating system or computer hardware? Additional hardware specific libraries are used to provide an interface between API and the hardware and between the user and the platform specific windowing system.

- a. DirectX
- b. Graphix Windowing Toolkit
- c. CGI
- d. OpenGL (Page 302)**

Question # 5 of 10 (Total Marks: 1) Select correct option:

When transforming a random Axis-Aligned Bounding Box defined by the points (nearx, neary, nearz) and (farx, fary, farz) to the standard orthographic viewing box, which affine transforms are used?

- a) shear and translation
- b) rotation and scale
- c) scale and shear
- d) translation and scale**

Question # 6 of 10 (Total Marks: 1) Select correct option:

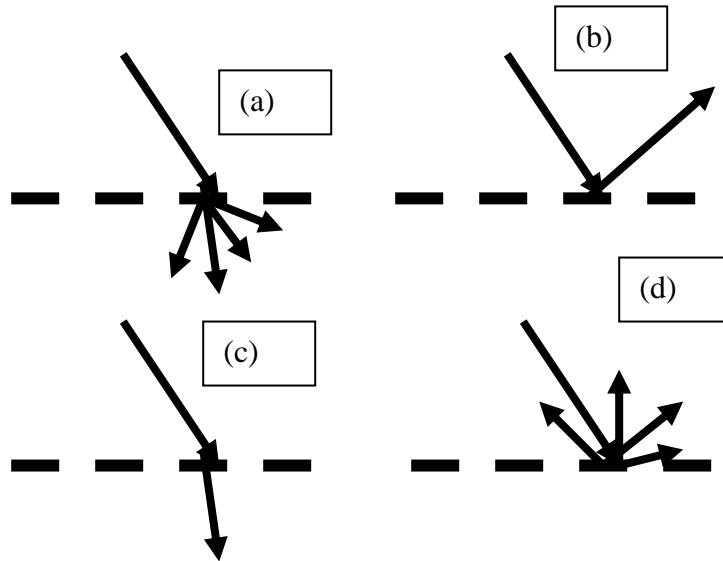
The _____ functions multiply the current matrix by a rotation matrix.

glRotated and glRotatef (Page 318)

Question # 7 of 10 (Total Marks: 1) Select correct option:

Match the pictures on the right with the corresponding term on the left. The arrows in the picture denote light rays. The dashed lines represent the material type to be considered. The key is in the interaction of the light rays with the material.

- Specular
- Diffuse
- Transparent
- Translucent



- [b,d,c,a](#) [Click here for detail](#)
- d,b,c,a
- a,c,b,d
- c,d,b,a

Solved by:

Well Wisher (Sahar)

Class

BSCS 6th Semester

Subject

Cs602 (computer graphics)

Solution Type:

Final Term Paper + Quizez

Institute:

Virtual University of Pakistan

1. Computer graphics and computer vision are _____ disciplines.

Related

Interrelated

Same

Different

Ref: Computer vision is, in some ways, the inverse of computer graphics. While computer graphics produces image data from 3D models, computer vision often produces 3D models from image data.

http://en.wikipedia.org/wiki/Computer_vision

2. OpenGL has become the industry's most widely used and supported _____ graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms.

Dimensional

Dimensional

2 Dimensional and 3-Dimensional

Ref: <http://www.opengl.org/about/>

3. _____ sets the reshape callback for the *current window*.

glutIdle function

glutKeyboardFunc

glutReshapeFunc

glutDisplayFunc

Ref:

<https://developer.apple.com/library/mac/documentation/Darwin/Reference/ManPages/man3/glutReshapeFunc.3.html>

4. _____ sets the global idle callback to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

glutIdle function

glutKeyboardFunc

glutReshapeFunc

glutDisplayFunc

Ref: <http://www.cs.uccs.edu/~ssemwal/man.html>

5. $X^2/a^2 - y^2/b^2 = 1$ is an equation of

Hyperbola **Page no : 70**

Parabola

None of given

Ellipse

6. To modify the object shape, shearing transformation cannot be used.

False **Page no : 124**

True

7. Rotation is performed around a fixed point called _____.

Pivot point rotation **Page no : 114**

8. Computer graphics is very helpful in producing graphical representations for scientific visualization and analysis

True

False

9. Save a line with both endpoints inside all clipping boundaries.

Trivial Reject

Trivial Accept **Page no : 137**

None of given

10. Dark lights are nothing more than lights in which one or more of the color values are _____.

Unknown

Negative **Page no : 230**

Positive

Zero

11. A series of _____ computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection.

Three

Page no : 366

Two

Four

Ten

At a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our _____ cells.

Eye

Retina

Cone

Page no : 393

12. This projection technique has the direction of projection perpendicular to the viewing plane, but the viewing direction is NOT perpendicular to one of the principle faces.

Orthographic Parallel Projection

Axonometric Parallel Projection **Page no : 189**

Oblique Parallel Projection

13. The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

Specular

Page no : 234

Ambient

Diffuse

14. In the Phong Reflection model, _____ light is the same everywhere.

a. **Ambient**

b. Diffuse

c. Specular

d. Emissive

e.

Ref: www.cs.unc.edu/~jwendt/classes/COMP136/quizzes/quiz3Answers.doc

15. A plane is two dimensional since in order to uniquely define any point on its surface we require _____ numbers.

Two

Three

Four

Five

16. In perspective projection, for your view to come out correctly, you will also want the _____ to pass through the middle of the screen.

X axis

Y axis

Z axis **Page no : 195**

None

17. Neither floating-point nor signed integer values are clamped to the range _____ before updating the current color.

0 , -1.0

-1 , 1

1 , -1

0, 1 **Page no : 316**

18. Bezier curve is the ideal standard for representing the _____ piecewise polynomial curves.

Most complex

Less complex

None of given

More complex **Page no : 333**

19. An object's _____ determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source.

Unit

Normal **Page no : 395**

None of given

20. Which was the most oldest shading model?

- a. **Flat Shading**
- b. Phong Shading
- c. Gouraud Shading

21. Which of the following affine transforms does NOT affect vectors?

- a) Scale
- b) Rotation
- c) Shear
- d) **Translation** **Page no : 113**

22) This projection technique does NOT have the direction of projection perpendicular to the viewing plane.

- a) Orthographic Parallel Projection
- b) Axonometric Parallel Projection
- c) **Oblique Parallel Projection** **Page no : 189**

23) This projection technique has the direction of projection perpendicular to the viewing plane, and the viewing direction is perpendicular to one of the principle faces.

- a) **Orthographic Parallel Projection** **Page no : 189**
- b) Axonometric Parallel Projection
- c) Oblique Parallel Projection

24) In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with a glRotatef function call?

- a) **GL_MODELVIEW** **Page no : 388**
- b) GL_PROJECTION

25) In OpenGL, there are several different matrices. We have discussed two of them in class. Which one of the below would be used in conjunction with glFrustum?

- a) GL_MODELVIEW
- b) **GL_PROJECTION** **Page no : 369**

26) Which of the following is the order that geometry operations are performed in OpenGL (where we read the order from left to right)?

- a) GL_PROJECTION → GL_MODELVIEW → Perspective division
- b) **GL_MODELVIEW → GL_PROJECTION → Perspective division**
- c) Perspective division → GL_PROJECTION → GL_MODELVIEW
- d) GL_MODELVIEW → Perspective division → GL_PROJECTION
- e) GL_PROJECTION → Perspective division → GL_MODELVIEW

27) The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Each piece of the Phong model uses different vectors and constants. Which portion does NOT include taking a dot product?

- a) **Ambient** **Page no : 234**
- b) Diffuse
- c) Specular

FINAL TERM PAPER 2010

Question No: 1 (Marks: 1) - Please choose one

Which of the following is NOT a modern application for Computer Graphics-----

- ▶ **Stop-motion animation** **Page no : 6**
- ▶ Computer Aided Geometric Design
- ▶ Video Games
- ▶ Scientific Visualization

Question No: 2 (Marks: 1) - Please choose one

Both Boundary Filling and Flood filling algorithms are non-recursive techniques,

- ▶ **False** **Page no : 97**
- ▶ True

Question No: 3 (Marks: 1) - Please choose one

TV series are made as simply as possible from the animation point of view. This approach is generally known as -----.

- ▶ Full animation
- ▶ **Limited animation** **Page no : 423**
- ▶ Low animation
- ▶ High resolution

Question No: 4 (Marks: 1) - Please choose one

An eight frame run cycle that ----- frame/frames to each step gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need dry brush or speed lines to make the movement flow.

- ▶ Two
- ▶ One
- ▶ Three

▶ **Four** **Page no :432**

Question No: 5 (Marks: 1) - Please choose one

Reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface.

- ▶ Forward scattering
- ▶ Diffuse Lambertian
- ▶ Backscattering

▶ **Retro** **Page no : 288**

Question No: 6 (Marks: 1) - Please choose one

What makes this really challenging to model is that the index of refraction for most materials is a function of the----- of the light. This means that not only is there a shift in the angle of refraction, but that the shift is different for differing -----of light.

- ▶ Reflecting angle, Reflecting angle
- ▶ Refracting angle, Refracting angle
- ▶ Frequency, Frequency

▶ **Wavelength, Wavelength** **Page no : 293**

Question No: 7 (Marks: 1) - Please choose one

The reflected light wave turns out to be a -----case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

- ▶ Abnormal
- ▶ Complex

▶ **Simple** **Page no : 291**

▶ Unknown

Question No: 8 (Marks: 1) - Please choose one

Tessellation can be adaptive to the _____ degree of curvature of a surface.

▶ **Local** **Page no : 165**

▶ Static

▶ Global

▶ Variable

Question No: 9 (Marks: 1) - Please choose one

_____ sets the reshape call back for the *current window*. The reshape callback is triggered when a window is reshaped.

▶ glutMainLoop

▶ glutIdleFunc

▶ **glutReshapeFunc** **Page no : 307**

▶ glutDisplayFunc

Question No: 10 (Marks: 1) - Please choose one

Signed integer colour components, when specified, are linearly mapped to floating-point values such that the most positive represent able value maps to 1.0, and the most negative represent able value maps to ----- . Floating-point values are mapped directly.

▶ -1.0

▶ 0.0

▶ 2.0

▶ **1.0** **Page no : 315**

Question No: 11 (Marks: 1) - Please choose one

NURBS stands for-----.

- ▶ Non Universal Rational Binary Spline
- ▶ Non Uniform Rational Binary Splines
- ▶ **Non Uniform Rational Beta Splines** Page no :320
- ▶ Non Universal Rational Beta Splines

Question No: 12 (Marks: 1) - Please choose one

An orthogonal set of vectors-----

- ▶ **Must be a set of linearly independent vectors**
- ▶ Must be a set of linearly dependent vectors
- ▶ Must be made up of the basis vectors (e1, e2, and e3)
- ▶ Can be made up of any set of vectors

Ref :

<http://www.ualberta.ca/dept/math/gauss/fcm/LinAlg/InRn/SbVctrSpc/OrthgnlLnrllyIndpndnt.htm>

Question No: 13 (Marks: 1) - Please choose one

Bezier curve is numerically the ----- of all the polynomial-based curves used in these applications.

- ▶ None of the given
- ▶ **Most stable**
- ▶ Less stable
- ▶ Most unstable

Ref :

<http://books.google.com.pk/books?id=YmQy799flPkC&pg=PA264&lpg=PA264&dq=Bezier+curve+is+numerically+the+-----+of+all+the+polynomial-based+curves+used+in+these+applications.&source=bl&ots=MHnr87FLlQ&sig=wG0oXl00vxWtEY7RnfNnOXlrc08&hl=en&sa=X&ei=nTroUaDdL6eB4ASioIHgBA&ved=0CCoQ6AEwAA#v=onepage&q=Bezier%20curve%20is%20numerically%20the%20----->

[%20of%20all%20the%20polynomial-based%20curves%20used%20in%20these%20applications.&f=false](#)

Question No: 14 (Marks: 1) - Please choose one

Bezier curve is the ideal standard for representing the ----- piecewise polynomial curves.

- ▶ None of the given
- ▶ Non complex
- ▶ Most complex
- ▶ **More complex** **repeated**

Question No: 15 (Marks: 1) - Please choose one

Keep polygon orientations consistent to make sure that when viewed from the outside, all the polygons on the surface are oriented in the ___ direction.

- ▶ None of the given
- ▶ Neither
- ▶ Different
- ▶ **Same** **Page no : 340**

Question No: 16 (Marks: 1) - Please choose one

The ----- is most simple example that exhibits the property self similarity.

- ▶ Mosse
- ▶ **Fern** **Page no : 350**
- ▶ None of the given
- ▶ Thohar

Question No: 17 (Marks: 1) - Please choose one

A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize ----- space.

- ▶ Multi-dimensional
- ▶ One-dimensional
- ▶ Two-dimensional
- ▶ **Three-dimensional** **Page no: 366**

Question No: 18 (Marks: 1) - Please choose one

Which of the following properties of rational Bezier curves fails if the weight assigned to a control point is negative?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ **Convex-Hull** **page no : 335**

Question No: 19 (Marks: 1) - Please choose one

In the Phong reflection model, there are 3 constants (a, b, c) which are used to describe the qualities of which of the following phenomena?

- ▶ **The attenuation of a point light source with distance** **Page no : 285**
- ▶ The size (in each dimension) which the light is assumed to have
- ▶ The amount to perturb reflection vectors as they are mirrored across the normal
- ▶ The material reaction to ambient, diffuse and specular light (respectively)

Question No: 20 (Marks: 1) - Please choose one

The Phong reflection model simplifies light-matter interactions into (essentially) 4 vectors and a number of constants. Which piece of the Phong model is responsible for giving spheres their bright white spots?

- ▶ **Specular** **repeated**
- ▶ Diffuse
- ▶ Ambient

Question No: 21 (Marks: 1) - Please choose one

When you hit a surface in ray tracing, generally shadow rays are tested against all objects in a scene. If these rays come back saying they hit an object in the scene, which of the following do you do?

- ▶ add all components (i.e. ambient, diffuse and specular) from that light source to the object.
- ▶ add all EXCEPT the ambient light from that light source to the object (i.e. diffuse and specular)
- ▶ **add only the ambient light from that light source to the object**
- ▶ add none of the light from that light source to the object

Question No: 22 (Marks: 1) - Please choose one

The Color Space tool is a handy tool that we can use to interactively add two colours together to see the effects of the various strategies for handling oversaturated colours.

- ▶ False
- ▶ **True** **page no : 230**

Question No: 23 (Marks: 1) - Please choose one

A polygon is usually defined by a sequence of ----- and Edges.

- ▶ Ending lines
- ▶ Points
- ▶ **Vertices** **Page no : 243**
- ▶ Edges

Question No: 24 (Marks: 1) - Please choose one

Which of the following properties of Bezier curves guarantees that a line passes through the control polygon as many times or more times than the line passes through the Bezier curve itself?

- ▶ End-point interpolation
- ▶ Variation Diminishing
- ▶ Symmetry
- ▶ **Convex-Hull**

Ref: <http://cagd.cs.byu.edu/~557/text/ch2.pdf>

Question No: 25 (Marks: 1) - Please choose one

Parity is a concept used to determine which _____ lie within a polygon. (Choose best suitable answer)

- ▶ Edge
- ▶ Vertices
- ▶ **Pixels** **Page no : 80**
- ▶ None of the given

Question No: 26 (Marks: 1) - Please choose one

The actual filling process in boundary filling algorithm begins when a point _____ of the figure is selected.

- ▶ **Outside the boundary**
- ▶ **Inside the boundary**
- ▶ At boundary
- ▶ None of the given

Question No: 27 (Marks: 1) - Please choose one

Weiler-Atherton Polygon Clipping technique modify the vertex-processing procedures for window boundaries so that _____ polygons are displayed correctly.

- ▶ Convex
- ▶ **Concave** **Page no : 245**
- ▶ Complex
- ▶ None of the given

Question No: 28 (Marks: 1) - Please choose one

If a line connecting any two points within a polygon does not intersect any edge, then it will be a _____ polygon.

- ▶ **Convex** **Page no : 78**
- ▶ Concave
- ▶ Complex
- ▶ None of the given

Question No: 29 (Marks: 1) - Please choose one

_____ can be defined as a mapping of point $P(x, y, z)$ onto its image $P'(x', y', z')$ in the view plane which constitutes the display surface.

- ▶ Mapping plane
- ▶ Three Coordinate Planes
- ▶ **View plane** **Repeated**
- ▶ Projection

Question No: 30 (Marks: 1) - Please choose one

The reflected light wave turns out to be a / an _____ case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now).

▶ Unknown

▶ **Simple** **Page no: 291**

▶ Complex

▶ Abnormal

Which of the following is NOT true about quaternions?

- a) They are made up of 4 numbers
- b) They should always be normalized to length 1
- c) They can be used to represent all affine transforms**
- d) They can be used to define the rotation of an object

FINAL TERM QUIZES

1) In class, we discussed three forms of shading for “Utah” graphics. Which was the first to use per vertex normals?

- a) Flat Shading
- b) Phong Shading
- c) Gouraud Shading** **Page no : 240**

2) Given any implicit equation, which of the following is true for all (x, y, z) that make the equation exactly zero?

- a) All those points are inside the object defined by the implicit equation
- b) All those points are on the surface of the object defined by the implicit equation** **Page no : 205**
- c) All those points are outside the object defined by the implicit equation
- d) You can't know anything without knowing what the implicit equation is

3) When solving ray-sphere intersections using the implicit equation for a sphere, you must solve the quadratic equation. Which of the following do you know if the B^2-4AC (i.e. the part under the square root) is negative?

- a) The ray intersects the sphere at a negative t... discard this result
- b) The ray intersects the sphere at a positive t... continue to the solution
- c) The ray does not intersect the sphere... discard this result** **Page no : 265**
- d) The ray begins inside the sphere... this is a special case

4)

_____ sets the global idle call back to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received.

Select correct option:

glutIdleFunc

glutMainLoop

glutDisplayFunc

glutReshapeFunc

Ref: <http://www.opengl.org/resources/libraries/glut/spec3/node63.html>

5)

A space curve can be confined to a plane.

Select correct option:

True

False Page no : 326

6)

A tangent vector certainly defines the slope at one end of the curve, but a vector has characteristics of.....

Select correct option:

direction

magnitude

both direction and magnitude Page no : 331

None of the given

7)

We allow the parametric variable to take on values only in the interval -----.

Select correct option:

$-1 \leq u \leq 0$

$0 \leq u \leq 2$

$0 \leq u \leq 1$ Page no : 321

$-1 \leq u \leq 1$

8)

The degree of a Bezier curve is equal to $n-1$, where n is the number of control points

Select correct option:

Yes **Page no : 334**

No

9)

Brenstien polynomial functions are the basic functions of _____ curves.

Select correct option:

NURBS

Bezier **Page no : 337**

Both NURBS and Bazier

None of the given

10)

A parametric curve is one whose defining equations are given in terms of a -----, common, independent variable called the parametric variable.

Select correct option:

Triple

Double

Single **Page no : 320**

None of the given

11)

Bit mask to select a window with multisampling support. If multisampling is not available, a ----- window will automatically be chosen.

Select correct option:

Non-multisampling **Page no : 305**

Multisampling

Mono-multisampling

Di-multisampling

12)

Bezier curve is tangent to the lines connecting _____.

Select correct option:

- First two points
- Last two points
- Fist two points and last two point**
- None of the given

13)

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers ----- information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

Select correct option:

Encapsulate **Page no : 297**

- Shows
- Hibernates
- None of the given

14)

A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are -----.

Select correct option:

- Binary polynomials
- Mono polynomials
- Quadratic polynomials

Cubic polynomials **Page no : 326**

15)

End points and an intermediate point on the curve, then we now ----- quantities that we can express in terms of these coefficients (3 points x 3 coordinates each), and we can use these three points to define a unique curve.

Select correct option:

- Six
- Three
- Two

Nine **Page no : 321**

16)

To convert the information in the A matrix into that required for the P matrix, we do some simple matrix algebra, First we have $UA=UNP$ then Simply $A = \text{-----}$

Select correct option:

UP

NP

UN

None of the given

17)

If we assign a different value to the parametric variable for the intermediate point, then we obtain different values for the coefficients. This, in turn, means that a different curve is produced, although it passes through the ----- three points.

Select correct option:

Isolate

Different

Same **Page no : 323**

None of the given

16)

In order to get a more realistic representation of lighting, we'll need to understand how light passes through a medium and how hitting the boundary layer at the ----- of two media can affect light's properties.

Select correct option:

Intersection **Page no : 295**

Union

Endpoints

Edges

17)

To ensure a smooth transition from one section of a piecewise _____ to the next, we can impose various continuity conditions at the connection points

Select correct option:

non parametric curve

parametric curve

polygon vector

Non of the these

Ref : www.mrl.snu.ac.kr/courses/CourseGraphics/Splines.ppt

18)

Bezier curve can represent the more complex piecewise _____ curve.

Select correct option:

Polynomial

Page no : 33

Exponential

Cubic

None of above

19)

Curve and surface equations can be expressed in either a parametric or a non parametric form.

Select correct option:

True

False

Page no : 333

20)

Using a lighting model based upon the Blinn Phong model means that we'll always get a uniform specular highlight based upon the colour of the ----- light and material, which means that all reflections based on this model, will be reminiscent of plastic.

Select correct option:

Union

Refracting

Page no : 291

Intersection

Reflecting

21)

If the current matrix (according to glMatrixMode) is multiplied by the translation matrix, with the product replacing the current matrix. That is, if M is the current matrix and T is the translation matrix, then M is replaced with -----.

Select correct option:

M-T

M+T

M/T

M*T

22)

With similar expressions for $y(u)$ and $z(u)$. Again the a , b , c and d terms are constant coefficients. As we did with Equation for a plane curve, we combine the $x(u)$, $y(u)$, and $z(u)$ expressions into a single vector equation $P(u) = \text{-----}$.

Select correct option:

Au^2+bu^1+cu+d

$Au^4+bu^3+cu^2+d^1$

$Au^3+bu^2+cu^2+d$

Au^3+bu^2+cu+d **Page no : 326**

23)

The matrix generated by `gluPerspective` is multiplied by the current matrix, just as if `glMultMatrix` were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to `gluPerspective` with a call to -----.

Select correct option:

`glRotated`

`gluPerspective`

`glTranslated`

`glLoadIdentity` **Page no : 313**

24)

Each number that makes up a matrix is called an _____ of the matrix.

- **Element** **Page no : 101**
- Variable
- Value
- Component

25)

Which one of the following step is not involved to write pixel using video BIOS services.

- Setting desired video mode
- Using BIOS service to set color of a screen pixel
- Calling BIOS interrupt to execute the process of writing pixel.
- **Using OpenGL service to set color of a screen pixel**

26)

Shadow mask methods can display a _____ range of colors.

- Small
- **Wide** **Page no : 29**
- Random
- Crazy

27)

Using Cohen-Sutherland line clipping, it is impossible for a vertex to be labeled 1111.

- **True**
- False

28)

Intensity of the electron beam is controlled by setting _____ levels on the control grid, a metal cylinder that fits over the cathode.

- Amplitude
- Current
- **Voltage** **Page no : 26**
- Electron



Lecture No.18 3D Transformations II

Rotation

Rotation is the process of moving a point in space in a non-linear manner. More particularly, it involves moving the point from one position on a sphere whose center is at the origin to another position on the sphere. Why would you want to do something like this? As we will show in later section, allowing the point of view to move around is only an illusion – projection requires that the POV be at the origin. When the user thinks the POV is moving, you are actually translating all your points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction; and when the user thinks the POV is looking down a new vector, you are actually rotating all the points in the opposite direction.

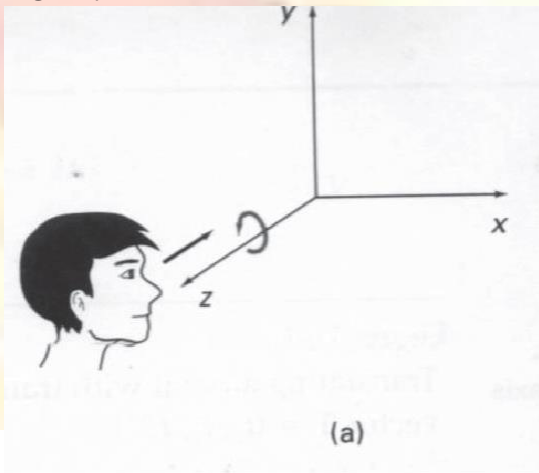
Normalization: Note that this process of moving your points so that your POV is at the origin looking down the +Z axis is called normalization.

Rotation a point requires that you know the coordinates for the point, and That you know the rotation angles.

You need to know three different angles: how far to rotate around the X axis(YZ rotation, or “pitch”); how far to rotate around the Y axis (XZ plane, or “yaw”); and how far to rotate around the Z axis (XY rotation, or “roll”). Conceptually, you do the three rotations separately. First, you rotate around one axis, followed by another, then the last. The order of rotations is important when you cascade rotations; we will rotate first around the Z axis, then around the X axis, and finally around the Y axis.

To show how the rotation formulas are derived, let’s rotate the point $\langle x,y,z \rangle$ around the Z axis with an angle of θ degrees.

ROLL:-



If you look closely, you should note that when we rotate around the Z axis, the Z element of the point does not change. In fact, we can just ignore the Z – we already know what it will be after the rotation. If we ignore the Z element, then we have the same case as if we were rotating the two-dimensional point $\langle x,y \rangle$ through the angle θ .

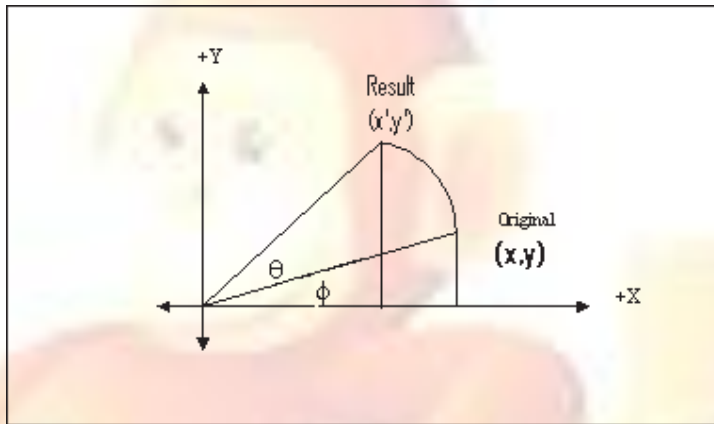
This is the way to rotate a 2-D point. For simplicity, consider the pivot at origin and rotate point P (x,y) where $x = r \cos\Phi$ and $y = r \sin\Phi$

If rotated by θ then:

$$\begin{aligned}x' &= r \cos(\Phi + \theta) \\ &= r \cos\Phi \cos\theta - r \sin\Phi \sin\theta\end{aligned}$$

and

$$\begin{aligned}y' &= r \sin(\Phi + \theta) \\ &= r \cos\Phi \sin\theta + r \sin\Phi \cos\theta\end{aligned}$$



Replacing $r \cos\Phi$ with x and $r \sin\Phi$ with y , we have:

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

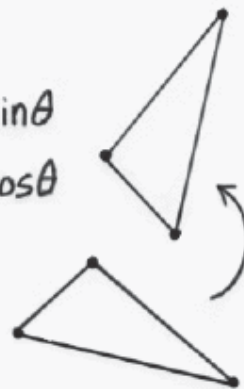
and

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

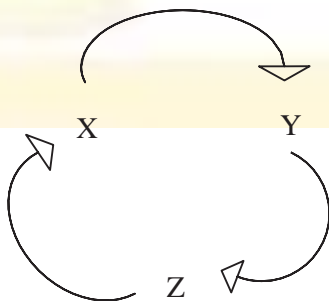
and

$$z' = z \text{ (as it does not change when rotating around z-axis)}$$

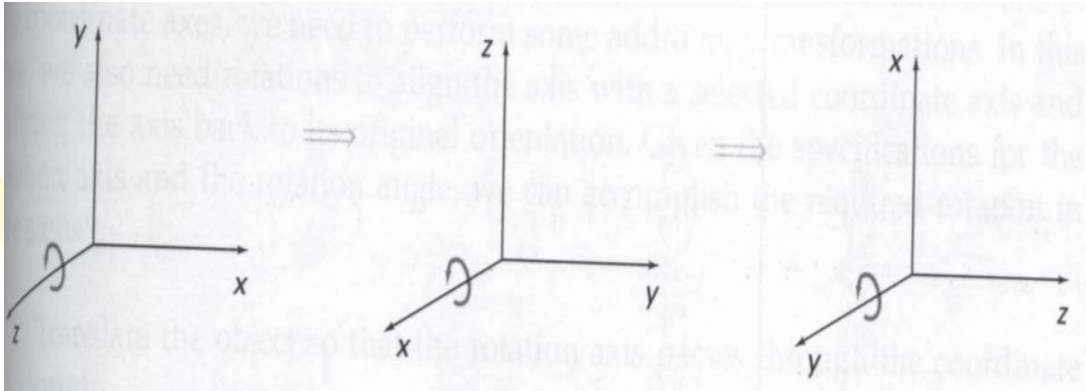
$$\begin{aligned}x' &= x \cos\theta - y \sin\theta \\ y' &= x \sin\theta + y \cos\theta\end{aligned}$$



Now for rotation around other axes, cyclic permutation helps form the equations for yaw and pitch as well:



In the above equations replacing x with y and y with z gives equations for rotation around x -axis. Now in the modified equations if we replace y with z and z with x then we get the equations for rotation around y -axis.



Rotation about x-axis (i.e. in yz plane):

$$\begin{aligned}x' &= x \\y' &= y \cos\theta - z \sin\theta \\z' &= y \sin\theta + z \cos\theta\end{aligned}$$

Rotation about y-axis (i.e. in xz plane):

$$\begin{aligned}x' &= z \sin\theta + x \cos\theta \\y' &= y \\z' &= z \cos\theta - x \sin\theta\end{aligned}$$

Using Matrices to create 3D

A matrix is usually defined as a two-dimensional array of numbers. However, I think you will find it much more useful to think of a matrix as an array of vectors. When we talk about vectors, what it really mean is an ordered set of numbers (a tuple in mathematics terms). We can use 3D graphics vectors and points interchangeably for this, since they are both 3-tuples (or triples).

In general we work with “square” matrices. This means that the number of vectors in the matrix is the same as the number of elements in the vectors that comprise it. Mathematically, we show a matrix as a 2-D array of numbers surrounded by vertical lines. For example:

$$\begin{array}{|c|c|c|} \hline x_1 & y_1 & z_1 \\ \hline x_2 & y_2 & z_2 \\ \hline x_3 & y_3 & z_3 \\ \hline \end{array}$$

we designate this as a 3*3 matrix (the first 3 is the number of rows, and the second 3 is the number of columns).

The “rows” of the matrix are the horizontal vectors that make it up; in this case, $\langle x_1, y_1, z_1 \rangle$, $\langle x_2, y_2, z_2 \rangle$, and $\langle x_3, y_3, z_3 \rangle$. In mathematics, we call the vertical vectors “columns.” In this case they are $\langle x_1, x_2, x_3 \rangle$, $\langle y_1, y_2, y_3 \rangle$ and $\langle z_1, z_2, z_3 \rangle$.

The most important thing we do with a matrix is to multiply it by a vector or another matrix. We follow one simple rule when multiplying something by a matrix: multiply each column by a multiplicand and store this as an element in the result. Now as I said

earlier, you can consider each column to be a vector, so when we multiply by a matrix, we are just doing a bunch of vector multiplies. So which vector multiply do you use-the dot product, or the cross product? You use the dot product.

We also follow on simple rule when multiplying a matrix by something: multiply each row by the multiplier. Again, rows are just vectors, and the type of multiplication is the dot product.

Let's look at some examples. First, let's assume that I have a matrix M, and I want to multiply it by a point $\langle x,y,z \rangle$, the first thing I know is that the vector rows of the matrix must contain three elements (in other words, three columns). Why? because I have to multiply those rows by my point using a dot product, and to do that, the two vectors must have the same number of elements. Since I am going to get dot product for each row in M, I will end up with a tuple that has one element for each row in M. as I stated earlier, we work almost exclusively with square matrices, since I must have three columns, M will also have three rows. Lets see:

$$\langle x,y,z \rangle * \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \{ \langle x,y,z \rangle * \langle 1,0,0 \rangle, \langle x,y,z \rangle * \langle 0,1,0 \rangle, \langle x,y,z \rangle * \langle 0,0,1 \rangle \} = \{ x,y,z \}$$

Using Matrices for Rotation

Roll (rotate about the Z axis):

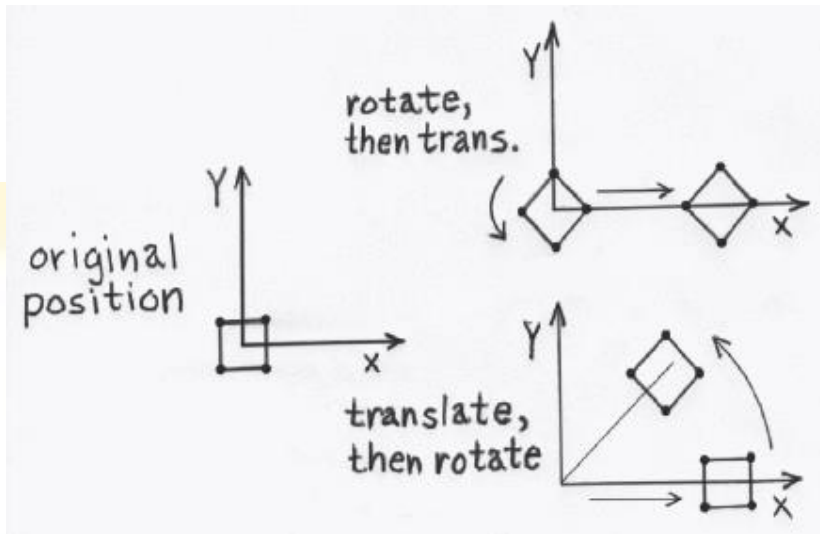
$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Pitch (rotate about the X axis):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Yaw (rotate about the Y axis):

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

**Example:**

To show this happening, let's manually rotate the point $\langle 2, 0, 0 \rangle$ 45 degrees clockwise about the z axis.

$$\mathbf{v}' = \mathbf{R}_z(45)\mathbf{v}$$

$$\mathbf{v}' = \begin{bmatrix} 0.707 & 0.707 & 0 \\ -0.707 & 0.707 & 0 \\ 0 & 0 & 1 \end{bmatrix} \times \begin{bmatrix} 2 \\ 0 \\ 0 \end{bmatrix}$$

$$\mathbf{v}' = \begin{bmatrix} 2 \times 0.707 + 0 \times 0.707 + 0 \times 0 \\ 2 \times -0.707 + 0 \times 0.707 + 0 \times 0 \\ 2 \times 0 + 0 \times 0 + 0 \times 0 \end{bmatrix}$$

$$\mathbf{v}' = \begin{bmatrix} 1.414 \\ -1.414 \\ 0 \end{bmatrix}$$

Now you can take an object and apply a sequence of transformations to it to make it do whatever you want. All you need to do is figure out the sequence of transformations needed and then apply the sequence to each of the points in the model.

As an example, let's say you want to rotate an object sitting at a certain point \mathbf{p} around its z axis. You would perform the following sequence of transformations to achieve this:

$$\mathbf{v} = \mathbf{vT}(-\mathbf{p})$$

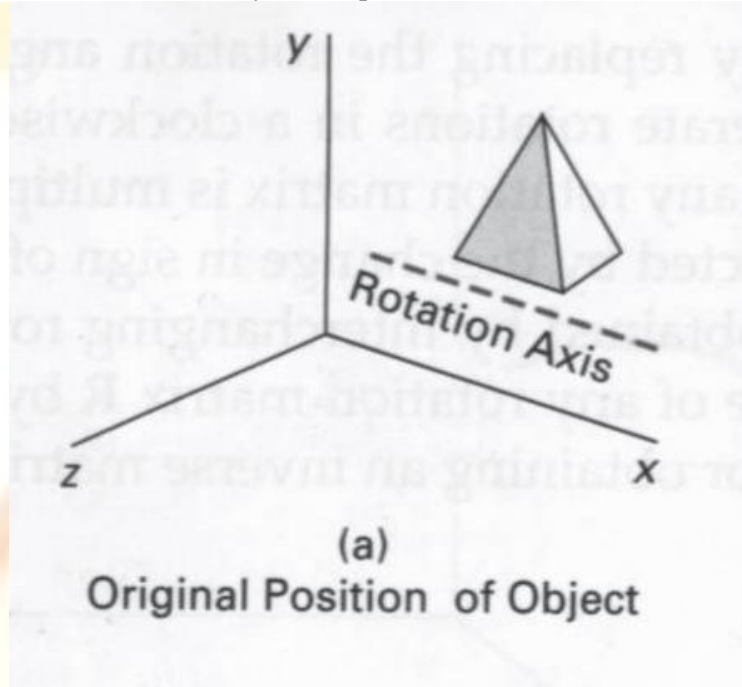
$$\mathbf{v} = \mathbf{vR}_z\left(\frac{\pi}{2}\right)$$

$$\mathbf{v} = \mathbf{vT}(\mathbf{p})$$

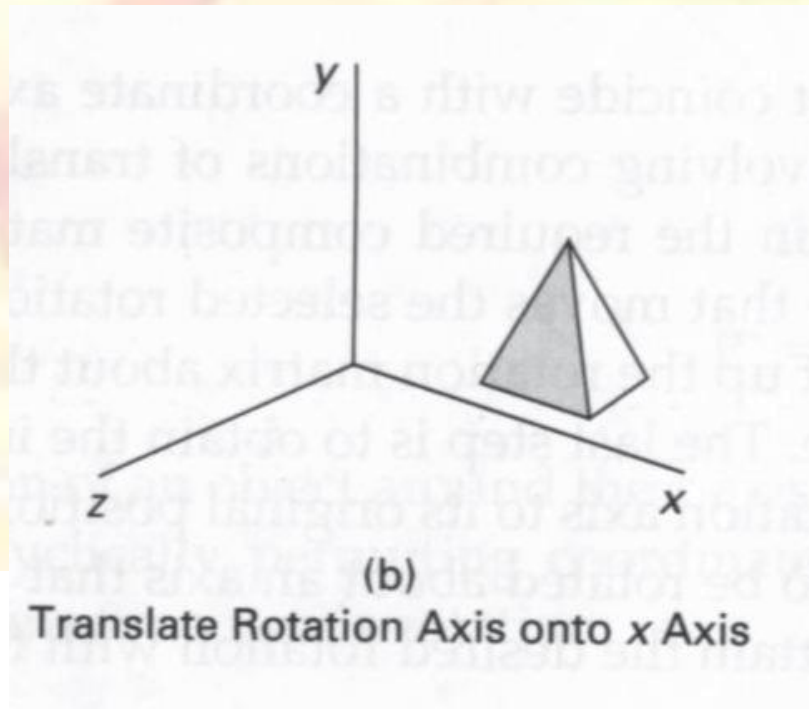
The first transformation moves a point such that it is situated about the world origin instead of being situated about the point \mathbf{p} . The next one rotates it (remember, you can only rotate about the origin, not arbitrary points in space). Finally, after the point is rotated, you want to move it back so that it is situated about \mathbf{p} . The final translation accomplishes this.

Rotation w.r.t. Arbitrary Axis:

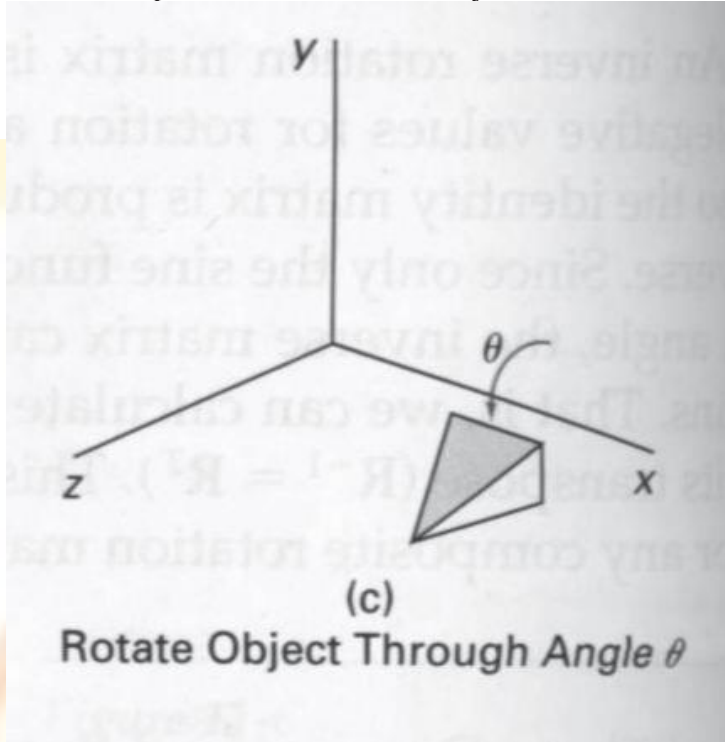
If an object is required to be rotated with respect to a line acting as an axis of rotation, arbitrarily, then the problem is addressed using multiple transformations. Let us assume that such an arbitrary axis is parallel to one of the coordinate axes, say x-axis.



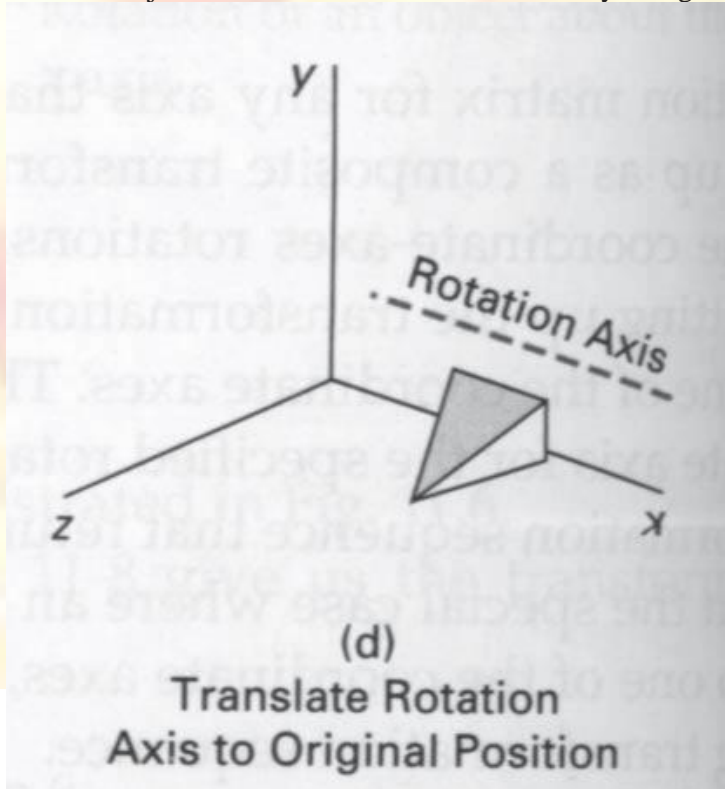
The first step in such case would be to translate the object such that the arbitrary axis coincides with the x-axis.



The next step would be to rotate the object w.r.t. x-axis through angle θ .

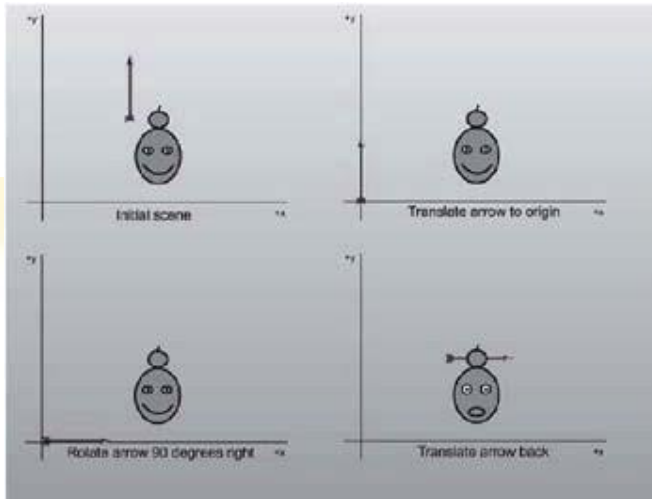


Then the object is translated such that the arbitrary axis gets back to its original position.

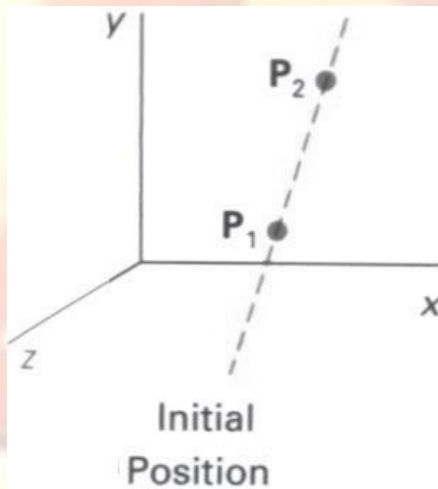


And thus the job is done.

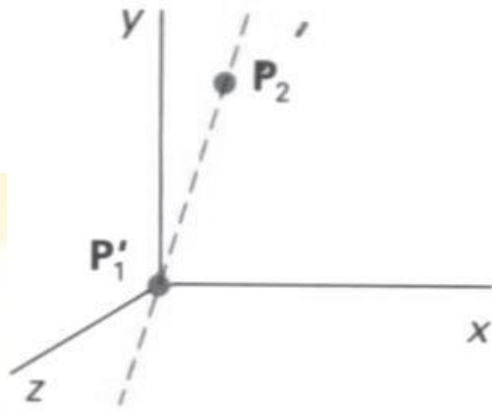
An interesting usage of compound transformations:-



Now, if the arbitrary axis is not parallel to any of the coordinate axes, then the problem is slightly more difficult. It only adds to the number of steps required to get the job done. Let P_1, P_2 be the line arbitrary axis.

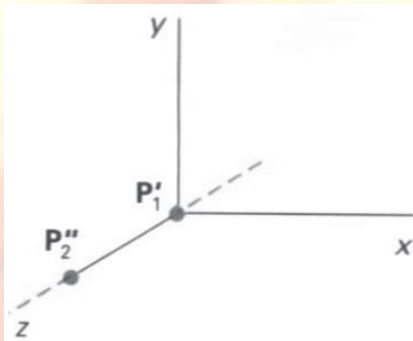


In the first step, the translation takes place that coincides the point P_1 to the origin. Points after this step are P_1' and P_2' .

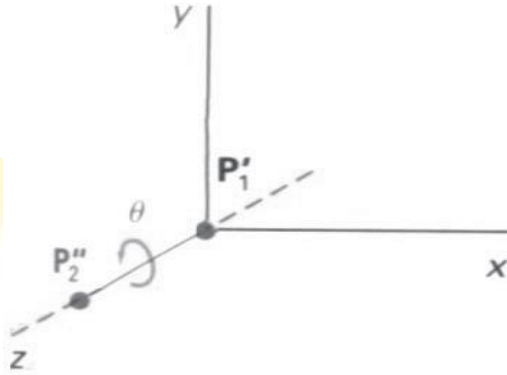


Step 1
Translate
 P_1 to the Origin

Now the arbitrary axis is rotated such that the point P_2' rotates to become P_2'' and lies on the z-axis.

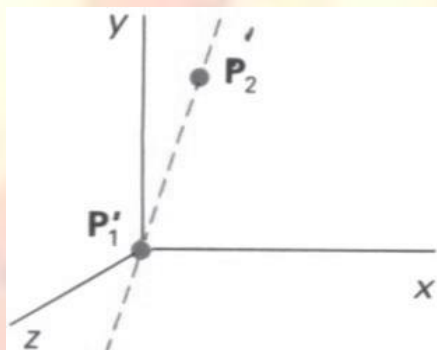


Step 2
Rotate P_2'
onto the z Axis



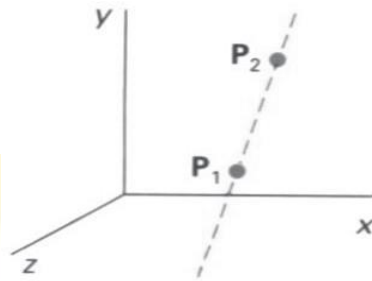
Step 3
Rotate the
Object Around the
z Axis

In the next step the object of interest is rotated around z-axis.



Step 4
Rotate the Axis
to the Original
Orientation

Now the object of interest is rotated about origin such that the arbitrary axis is poised like in above figure. Point P_2'' gets back to its previous position P_2' .



Step 5
Translate the
Rotation Axis
to the Original
Position

Finally the translation takes place to position the arbitrary axis back to its original position.

Scaling

Coordinate transformations for scaling relative to the origin are

$$X' = X \cdot S_x$$

$$Y' = Y \cdot S_y$$

$$Z' = Z \cdot S_z$$

Scaling an object with transformation changes the size of the object and reposition the object relative to the coordinate origin. If the transformation parameters are not all equal, relative dimensions in the object are changed.

Uniform Scaling : We preserve the original shape of an object with a uniform scaling ($S_x = S_y = S_z$)

Differential Scaling : We do not preserve the original shape of an object with a differential scaling ($S_x \neq S_y \neq S_z$)

Scaling relative to the coordinate Origin:

Scaling transformation of a position $P = (x, y, z)$ relative to the coordinate origin can be written as

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling with respect to a selected fixed position:

Scaling with respect to a selected fixed position (X_f, Y_f, Z_f) can be represented with the following transformation sequence:

Translate the fixed point to the origin.

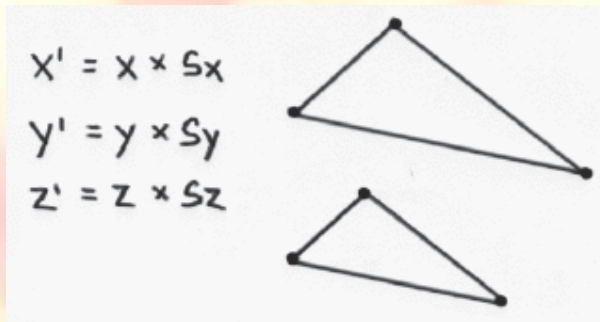
Scale the object relative to the coordinate origin

Translate the fixed point back to its original position

For these three transformations we can have composite transformation matrix by multiplying three matrices into one

$$\begin{bmatrix} 1 & 0 & 0 & X_f \\ 0 & 1 & 0 & Y_f \\ 0 & 0 & 1 & Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -X_f \\ 0 & 1 & 0 & -Y_f \\ 0 & 0 & 1 & -Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} S_x & 0 & 0 & (1-S_x)X_f \\ 0 & S_y & 0 & (1-S_y)Y_f \\ 0 & 0 & S_z & (1-S_z)Z_f \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



Reflection

A three-dimensional reflection can be performed relative to a selected reflection axis or with respect to a selected reflection plane. In general, three-dimensional reflection matrices are set up similarly to those for two dimensions. Reflections relative to a given axis are equivalent to 180 degree rotations.

The matrix representation for this reflection of points relative to the X axis

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix representation for this reflection of points relative to the Y axis

$$\begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The matrix representation for this reflection of points relative to the xy plane is

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shears

Shearing transformations can be used to modify object shapes.

As an example of three-dimensional shearing, the following transformation produces a z-axis shear:

$$\begin{bmatrix} 1 & 0 & a & 0 \\ 0 & 1 & b & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Parameters a and b can be assigned any real values. The effect of this transformation matrix is to alter x and y- coordinate values by an amount that is proportional to the z value, while leaving the z coordinate unchanged.

y-axis Shear

$$\begin{bmatrix} 1 & a & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & c & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

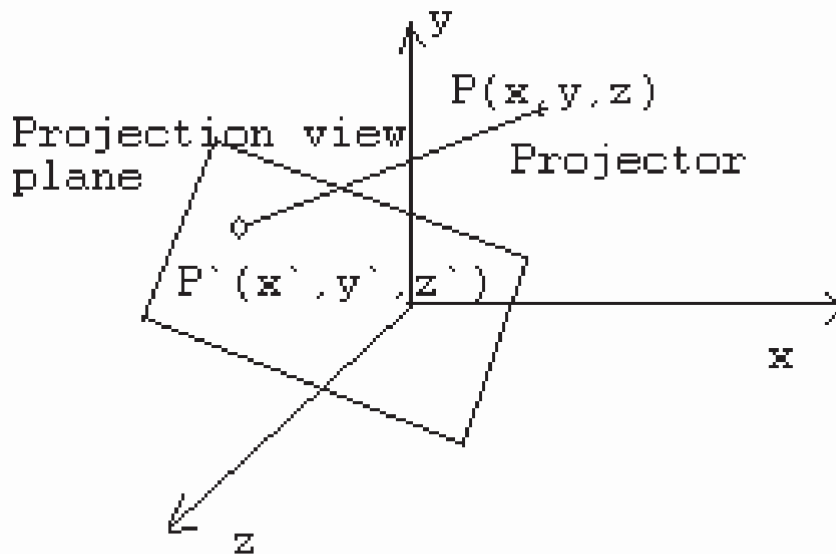
x-axis Shear

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ b & 1 & 0 & 0 \\ c & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Lecture No.19 Projections

For centuries, artists, engineers, designers, drafters, and architects have been facing difficulties and constraints imposed by the problem of representing a three-dimensional object or scene in a two-dimensional medium -- the problem of projection. The implementers of a computer graphics system face the same challenge.

Projection can be defined as a mapping of point $P(x,y,z)$ onto its image $P'(x',y',z')$ in the projection plane or view plane, which constitutes the display surface. The mapping is determined by a projection line called the projector that passes through P and intersects the view plane.



The Problem of Projection

There are two basic methods of projection

Parallel Projection
Perspective Projection

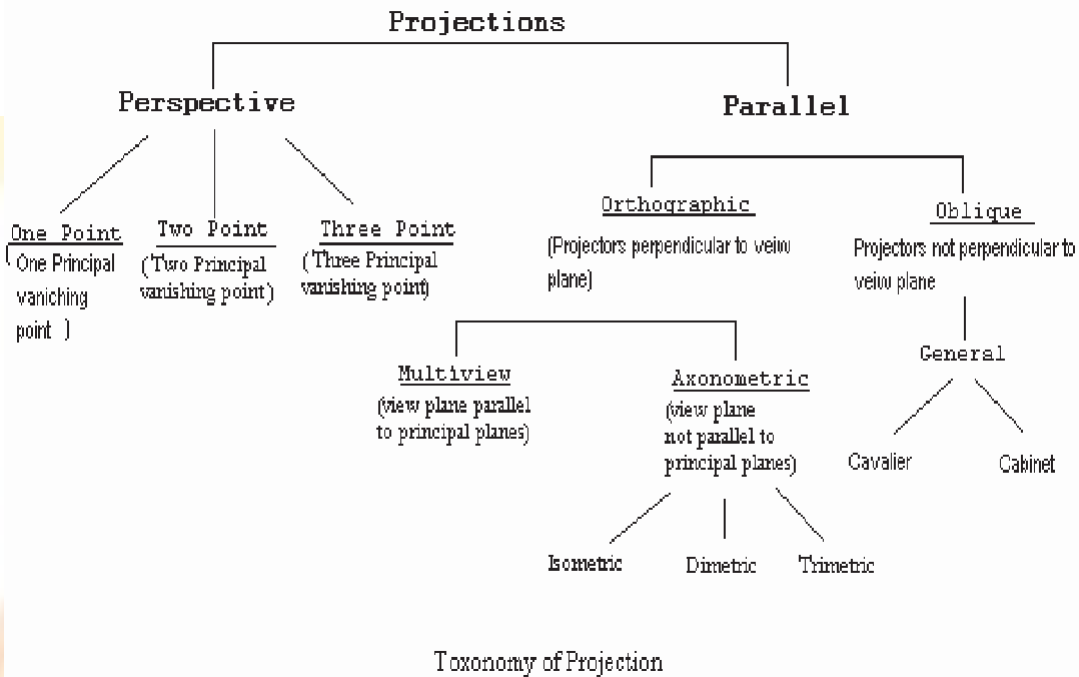
These methods are used to solve the basic problems of pictorial representations

We characterize each method and introduce the mathematical description of the projection process respectively.

Taxonomy of Projection

We can construct different projections according to the view that is desired.

Following figure provides taxonomy of the families of perspective and parallel projections. Some projections have names – cavalier, cabinet, isometric, and so on. Other projections qualify the main type of projection – one principal vanishing–point perspective and so forth.



Parallel Projection

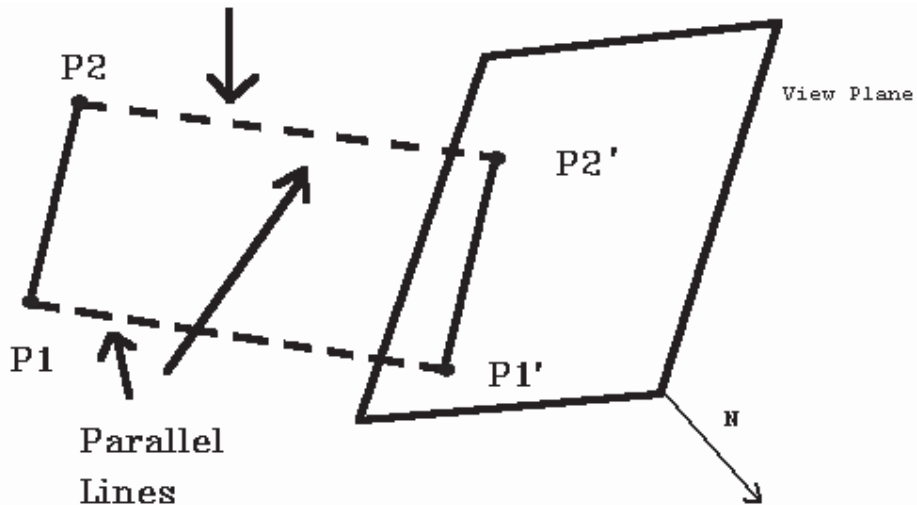
Parallel projection methods are used by drafters and engineers to create working drawings of an object which preserves its scale and shape. The complete representation of these details often requires two or more views (projections) of the object onto different view planes.

In parallel projection, image points are found as the intersection of the view plane with a projector drawn from the object point and having a fixed direction. The direction of projection is the prescribed direction for all projections. *Orthographic projections are characterized by the fact that the direction of projection is perpendicular to the view plane.* When the direction of projection is parallel to any of the principal axes, this produces the front, top, and side views of mechanical drawings (also referred to as multi view drawings).

Axonometric projections are orthographic projections in which the direction of projection is not parallel to any of the three principal axes. *Non orthographic parallel projections are called oblique parallel projection.*

Mathematical Description of a Parallel Projection

Projection rays (projectors) emanate from a Center of Projection (COP) and intersect Projection Plane (PP). The COP for parallel projectors is at infinity. The length of a line on the projection plane is the same as the "true Length".



There are two different types of parallel projections:

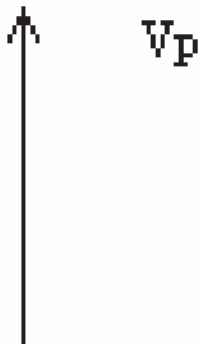
Orthographic

Oblique

1) Orthographic Projection

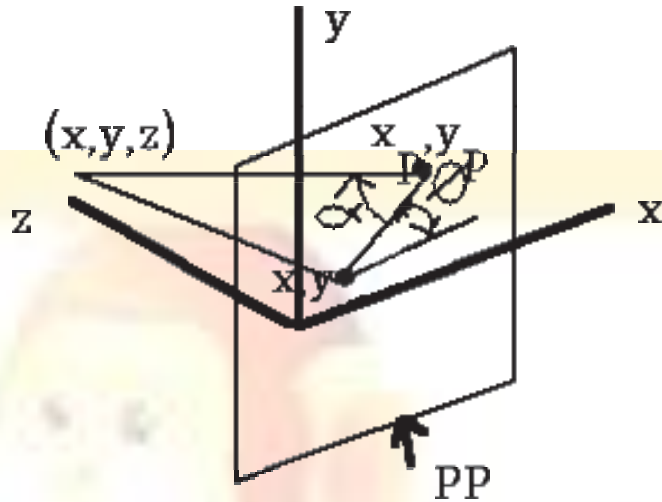
If the direction of projection is perpendicular to the projection plane then it is an **orthographic** projection.

Projection plane



Orthographic projection

Look at the parallel projection of a point (x, y, z) . (Note the left handed coordinate system). The projection plane is at $z = 0$. x, y are the orthographic projection values and x_p, y_p are the oblique projection values (at angle α with the projection plane)

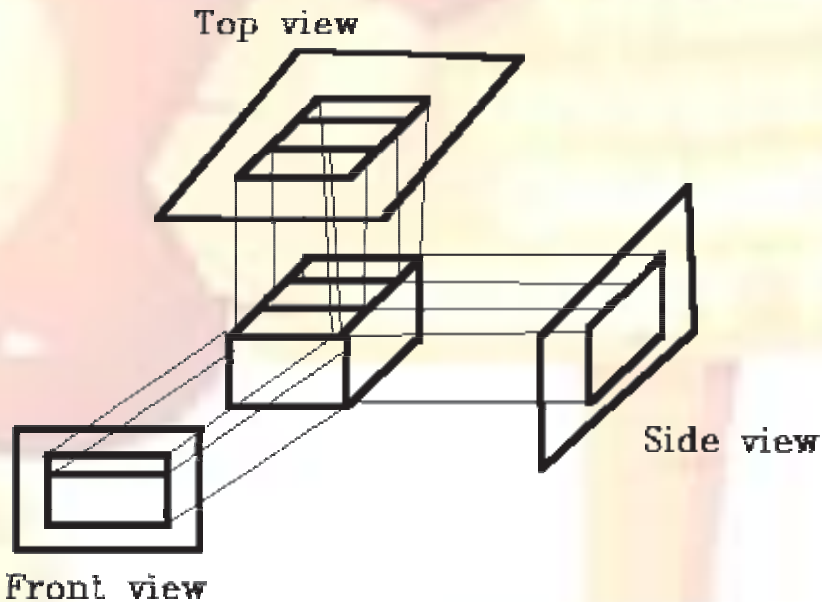


Look at orthographic projection: it is simple, just discard the z coordinates. Engineering drawings frequently use front, side, top orthographic views of an object.

Axonometric orthographic projection

Orthographic projections that show more than one side of an object are called **axonometric** orthographic projections.

Here are three orthographic views of an object.



There are three axonometric projections

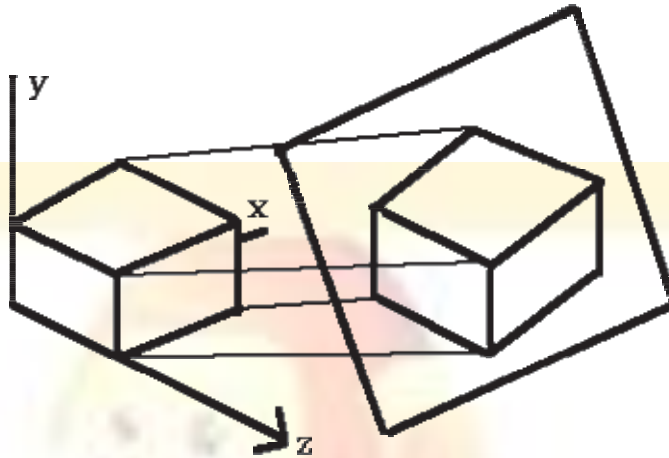
isometric

Dimetric

Trimetric

1) Isometric

The most common axonometric projection is an **isometric** projection where the projection plane intersects each coordinate axis in the model coordinate system at an equal distance or the direction of projection makes equal angles with all of the three principal axes



The projection plane intersects the x , y , z axes at equal distances and the projection plane Normal makes an equal angle with the three axes.

To form an orthographic projection $x_p = x$, $y_p = y$, $z_p = 0$. To form different types e.g., Isometric, just manipulate object with 3D transformations.

2) Dimetric

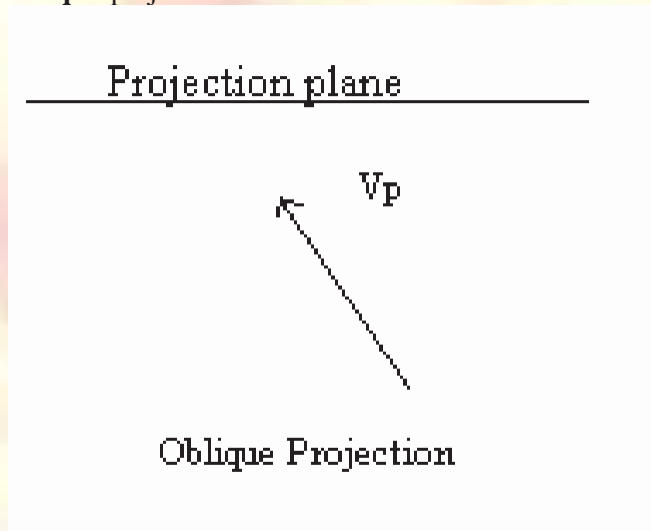
The direction of projection makes equal angles with exactly two of the principal axes

3) Trimetric

The direction of projection makes unequal angles with the three principal axes

Oblique Projection

If the direction of projection is not perpendicular to the projection plane then it is an **oblique** projection.



The projectors are not perpendicular to the projection plane but are parallel from the object to the projection plane.

Transformation equations for an orthographic parallel projection are straightforward. If the view plane is placed at position Z_{vp} along the Z axis, Then any point (x,y,z) in viewing coordinates is transformed to projection coordinates as:

$$X_p = x$$

$$Y_p = y$$

Where the original Z-coordinate value is preserved for the depth information needed in depth cueing and visible-surface determination procedures.

An oblique projection is obtained by projecting points along parallel lines that are not perpendicular to the projection plane. In some applications packages, an oblique projection vector is specified with two angles, alpha and phi, as shown in the figure. Point (x,y,z) is projected to position (X_p, Y_p) on the view plane. Orthographic projection coordinates on the plane are (x,y) . The oblique projection line from (x,y,z) to (X_p, Y_p) makes an angle alpha with the line on the projection plane that joins (X_p, Y_p) and (x, y) . This line, of length L, is at an angle phi with the horizontal direction in the projection plane. We can express the projection coordinates in terms of x, y, L, and phi as

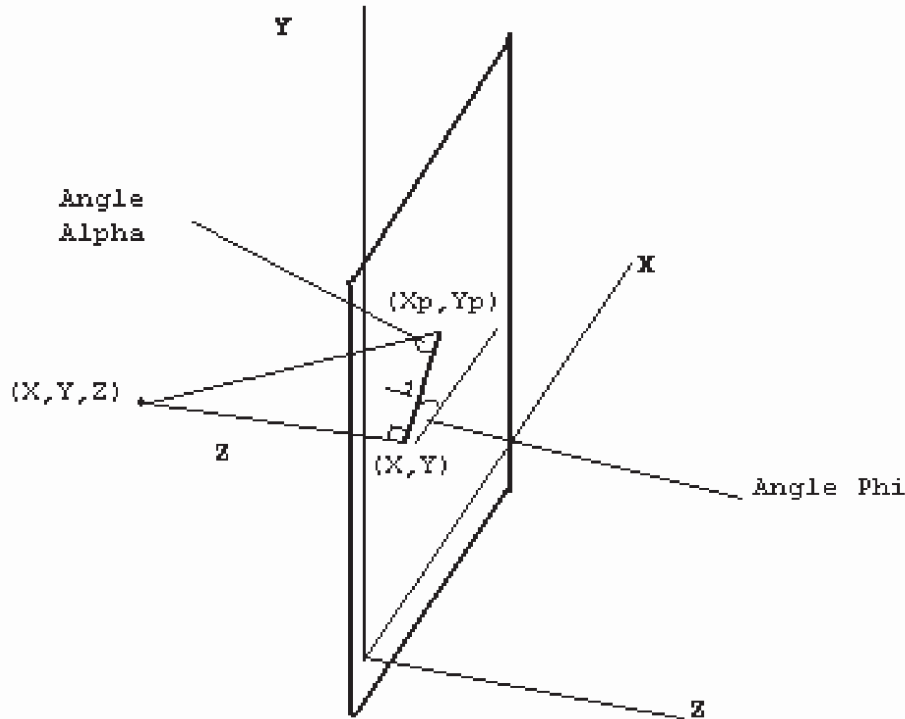


Figure: Oblique Projection of coordinate position (x,y,z) to position (X_p, Y_p) on the view plane

$$\cos(\phi) = X_p - x / L$$

$$\sin(\phi) = Y_p - y / L$$

$$X_p = x + L \cos(\phi)$$

$$Y_p = y + L \sin(\phi)$$

Length L depends on the angle alpha and the z coordinate of the point to be projected:

$$\tan(\alpha) = z / L$$

Thus,

$$L = z * 1 / \tan(\alpha)$$

$$L = z * L1$$

Where $L1$ is the inverse of $\tan(\alpha)$, which is also the value of L when $z = 1$, we can then write the oblique projection equations.

$$X_p = x + z (L1 \cos(\phi))$$

$$Y_p = y + z (L1 \sin(\phi))$$

The transformation matrix for producing any parallel projection onto the xy plane can be written as

$$\text{Parallel M} = \begin{bmatrix} 1 & 0 & L1 \cos(\phi) & 0 \\ 0 & 1 & L1 \sin(\phi) & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Now if $\text{Alpha} = 90^\circ$ (projection line is perpendicular to Projection Plane) then $\tan(\text{Alpha}) = \text{infinity} \Rightarrow L1 = 0$, so have an orthographic projection.

Two special cases of oblique projection

Cavalier

Cabinet

1) Cavalier

$\text{Alpha} = 45^\circ$, $\tan(\text{Alpha}) = 1 \Rightarrow L1 = 1$ this is a **Cavalier** projection such that all lines perpendicular to the projection plane are projected with no change in length.



2) Cabinet

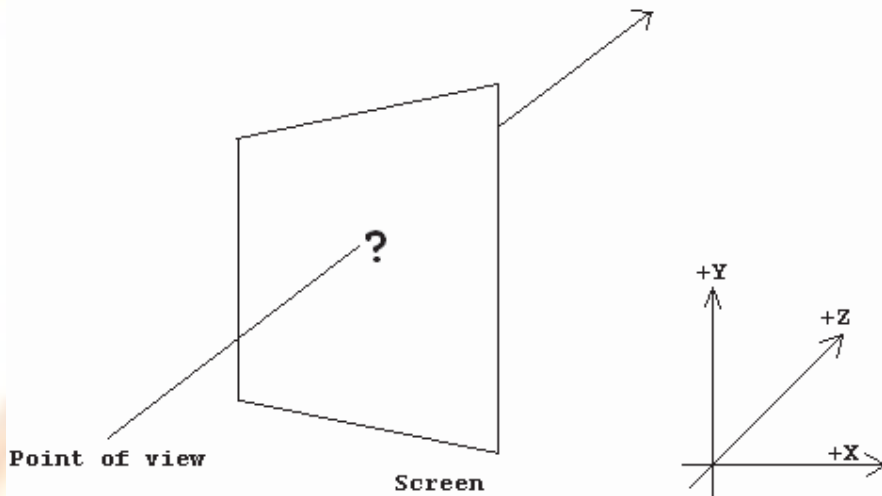
$\tan(\text{Alpha}) = 2$, $\text{Alpha} = 63.40^\circ$, $L1 = 1/2$

Lines which are perpendicular to the projection plane are projected at $1/2$ length. This is a **Cabinet** projection



Lecture No.20 Perspective Projection

Now that you have a structure that can store a three dimensional point (Point3D), how do you calculate the corresponding screen pixel? First, let's look at what you are modeling. Following figure shows how it would look.



This is a mathematical task as pictured. However, we can make it much simpler if we impose the following requirements:

- the point of View (POV) must lie on the Z axis, and
- the screen plane must be parallel to the X-Y plane,
- with the left and right edges of the screen parallel to the Y axis, and
- the top and bottom edges of the screen parallel to the X axis,
- for your view to come out correctly, you will also want the Z axis to pass through the middle of the screen.

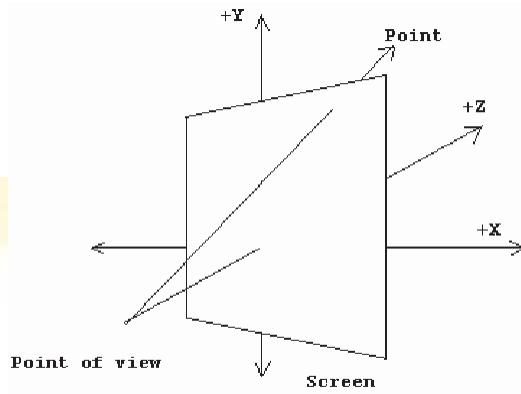
Why?

The POV represents the viewer's eye, and we presume that the viewer will be behind the center of the screen.

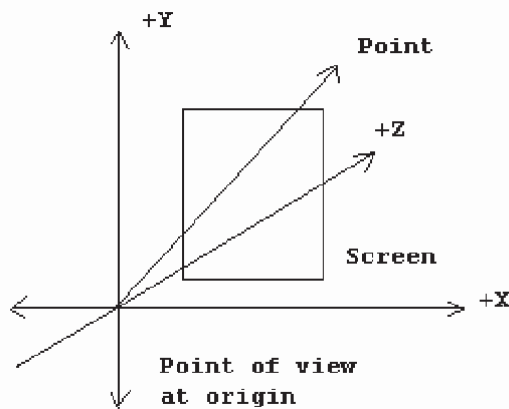
Note: We will use Left hand rule to describe 3D coordinate system.

Two common approaches are used with this;

1. The first approach is where the POV is at some point $(0, 0, -z)$ and the screen lies on the X-Y plane, graphically, this looks like figure given below:

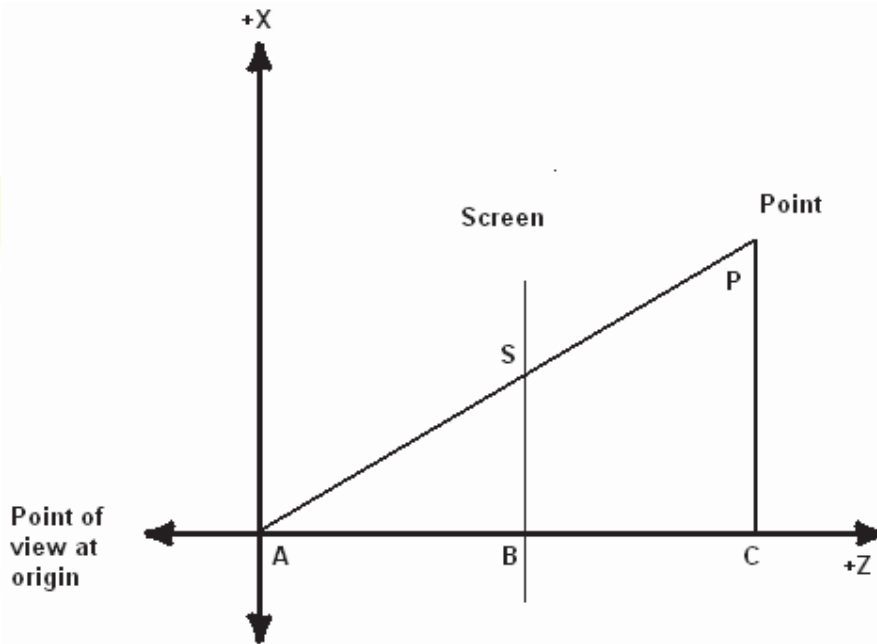


2. The second approach is where the POV lies at the origin, and the screen lies on a plane at some +z coordinate, as shown in figure given below:



As we will see later, this second approach is much more convenient when we add features making it possible for the POV to move around the 3D world or for objects to move around in the world.

Calculating the screen pixel that correlates to a 3D point is now a matter of simple geometry. From a viewpoint above the screen and POV (looking at the X-Z plane), the geometry appears like the one shown in figure below:



In geometric terms, we say that the triangle from A to B to S is similar to the triangle from A to C to P because the three angles that make up the triangles are the same: the angle from AB to AS is the same as the angle from AC to AP, the two right angles are both 90 degrees, and therefore the remaining two angles are the same (the sum of the angles in a triangle is always 180 degrees). What also holds true from similar triangles is that the ratio of two sides holds between the similar triangles; this means that the ratio of BS to AB is the same as the ratio of CP to AC. But we know what AB is—it is Screen.z! and we know what AC is—it is point.z! and we know what CP is—it is point.x! Therefore:

$$|BS| / |AB| = |CP| / |AC|$$

$$|BS| = |AB| * |CP| / |AC|$$

$$|BS| = \text{Screen.z} * \text{point.x} / \text{point.z}$$

Screen.z is the distance d from the point of view at origin or the scaling factor.

Notice that |BS| is the length of the line segment that goes from B to S in world units. But we normally address the screen with the point (0,0) at the top left, with +X pixels moving to the right, and +Y pixels moving down—and not from the middle of the screen. And we draw to the screen in pixel units – not our world units (unless, of course, 1.0 in your world represents one pixel).

There is a final transformation that the points must go through in the transformation process. This transformation maps 3D points defined with respect to the view origin (in view space) and turns them into 2D points that can be drawn on the display. After transforming and clipping the polygons that make up the scene such that they are visible on the screen, the final step is to move them into 2D coordinates, since in order to actually draw things on the screen you need to have absolute x, y coordinates on the screen to draw.

The way this used to be done was without matrices, just as an explicit projection calculation. The point (x,y,z) would be mapped to (x', y') using the following equations

$$x' = \text{scale} \frac{x}{z} + x_{\text{Center}}$$

$$y' = \text{height} - \left(\text{scale} \frac{y}{z} + y_{\text{Center}} \right)$$

Where x_{Center} and y_{Center} were half of the width and height of the screen respectively, these days more complex equations are used, especially since there is now the need to make provisions for z-buffering. While you want x and y to still behave the same way, you don't want to use a value as arbitrary as scale .

Instead, a better value to use in the calculation of the projection matrix is the horizontal field of view (fov). The horizontal fov will be hard coded, and the code chooses a vertical field of view that will keep the aspect ratio of the screen. This makes sense: You couldn't get away with using the same field of view for both horizontal and vertical directions unless the screen was square; it would end up looking vertically squashed.

Finally, you also want to scale the z values appropriately. In future, We'll teach you about z-buffering, but for right now just make note of an important feature: They let you clip out certain values of z -range. Given the two variables z_{near} and z_{far} , nothing in front of z_{near} will be drawn, nor will anything behind z_{far} . To make the z -buffer work swimmingly on all ranges of z_{near} and z_{far} , you need to scale the valid z values to the range of 0.0 to 1.0.

The Perspective Projection Matrix

The aspect ratio of screen, width and height is calculated as

$$\text{aspect} = \frac{\text{height}}{\text{width}}$$

$$w = \text{aspect} \frac{\cos(\text{fov})}{\sin(\text{fov})}$$

$$h = \frac{\cos(\text{fov})}{\sin(\text{fov})}$$

$$q = \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}}$$

With these parameters, the following projection matrix can be made:

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & q & 1 \\ 0 & 0 & -q(z_{\text{near}}) & 0 \end{bmatrix}$$

Just for a sanity check, check out the result of this matrix multiplication:

$$[x \ y \ z \ 1] \begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & q & 1 \\ 0 & 0 & -q(z_{\text{near}}) & 0 \end{bmatrix} = [wx \ hy \ qz - q(z_{\text{near}}) \ z]$$

This is almost the result wanted, but there is more work to be done. Remember that in order to extract the Cartesian (x,y,z) coordinates from the vector, the homogenous w component must be 1.0. Since, after the multiplication, it's set to z (which can be any value), all four components need to be divided by z to normalize it (and have the homogeneity factor equal 1). This gives the following Cartesian coordinate:

$$\left[\frac{wx}{z} \quad \frac{hy}{z} \quad q \left(1 - \frac{z_{near}}{z} \right) \quad 1 \right]$$

As you can see, this is exactly what was wanted. The width and height are still scaled by values as in the above equation and they are still divided by z . The visible x and y pixels are mapped to $[-1,1]$, so before rasterization Application multiplies and adds the number by $xCenter$ or $yCenter$. This, in essence, maps the coordinates from $[-1,1]$ to $[0,width]$ and $[0,height]$.

With this last piece of the puzzle, it is now possible to create the entire transformation pipeline. When you want to render a scene, you set up a world matrix (to transform an object's local coordinate points into world space), a view matrix (to transform world coordinate points into a space relative to the viewer), and a projection matrix (to take those viewer-relative points and project them onto a 2D surface so that they can be drawn on the screen). You then multiply the world, view, and projection matrices together (in that order) to get a total matrix that transforms points from object space to screen space.

$$\begin{aligned} \mathbf{v}_{world} &= \mathbf{v}_{local} \mathbf{M}_{world} \\ \mathbf{v}_{view} &= \mathbf{v}_{world} \mathbf{M}_{view} \\ \mathbf{v}_{screen} &= \mathbf{v}_{view} \mathbf{M}_{projection} \\ \mathbf{v}_{screen} &= \mathbf{v}_{local} (\mathbf{M}_{world} \mathbf{M}_{view} \mathbf{M}_{projection}) \end{aligned}$$

To draw a triangle, for example, you would take its local space points defining its three corners and multiply them by the transformation matrix. Then you have to remember to divide through by the w component. The points are now in screen space and can be filled in using a 2D raster algorithm. Drawing multiple objects is a snap, too. For each object in the scene all you need to do is change the world matrix and reconstruct the total transformation matrix.

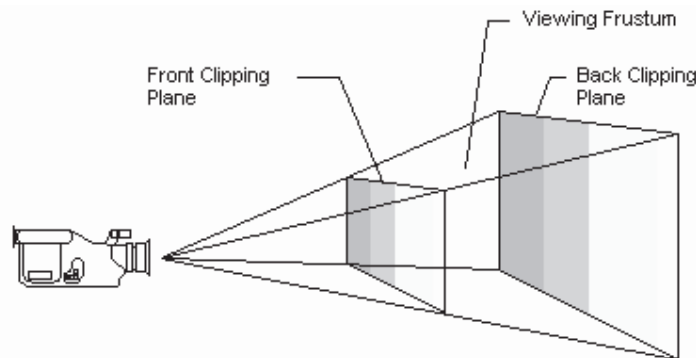
The Perspective Projection Matrix Used by Microsoft Direct3D

The projection matrix is typically a scale and perspective projection. The projection transformation converts the viewing frustum into a cuboid shape. Because the near end of the viewing frustum is smaller than the far end, this has the effect of expanding objects that are near to the camera; this is how perspective is applied to the scene.

The Viewing Frustum

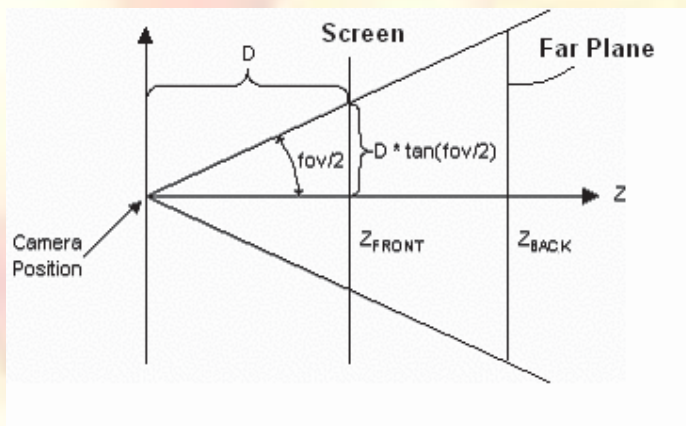
A viewing frustum is 3-D volume in a scene positioned relative to the viewport's camera. The shape of the volume affects how models are projected from camera space onto the screen. The most common type of projection, a perspective projection, is responsible for making objects near the camera appear bigger than objects in the distance. For perspective viewing, the viewing frustum can be visualized as a pyramid, with the camera

positioned at the tip. This pyramid is intersected by a front and back clipping plane. The volume within the pyramid between the front and back clipping planes is the viewing frustum. Objects are visible only when they are in this volume.



If you imagine that you are standing in a dark room and looking through a square window, you are visualizing a viewing frustum. In this analogy, the near clipping plane is the window, and the back clipping plane is whatever finally interrupts your view—the skyscraper across the street, the mountains in the distance, or nothing at all. You can see everything inside the truncated pyramid that starts at the window and ends with whatever interrupts your view, and you can see nothing else.

The viewing frustum is defined by *fov* (field of view) and by the distances of the front and back clipping planes, specified in *z*-coordinates.



In this illustration, the variable D is the distance from the camera to the origin of the space that was defined in the last part of the geometry pipeline—the viewing transformation. This is the space around which you arrange the limits of your viewing frustum. For information about how this D variable is used to build the projection matrix

The Matrix

In *the viewing frustum*, the distance between the camera and the origin of the viewing transformation space is defined arbitrarily as D , so the projection matrix looks like:

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

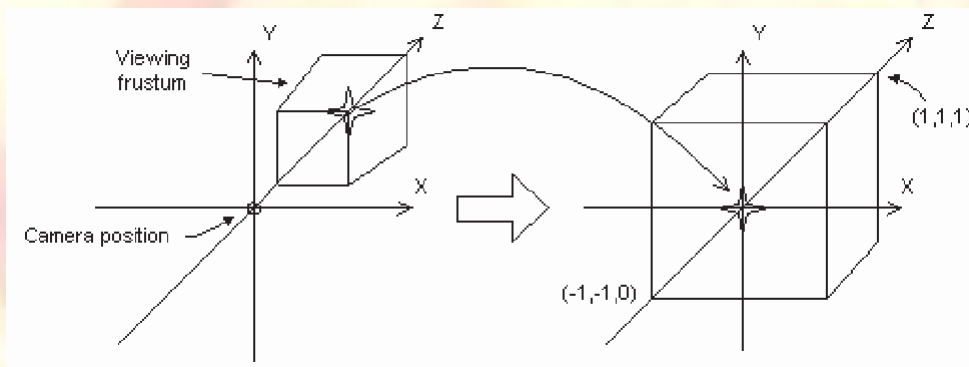
The viewing matrix translates the camera to the origin by translating in the z direction by $-D$. The translation matrix is as follows:

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -D & 1 \end{bmatrix}$$

Multiplying the translation matrix by the projection matrix ($T*P$) gives the composite projection matrix. It looks like:

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

The following illustration shows how the perspective transformation converts a viewing frustum into a new coordinate space. Notice that the frustum becomes cuboid and also that the origin moves from the upper-right corner of the scene to the center.



In the perspective transformation, the limits of the x- and y-directions are -1 and 1. The limits of the z-direction are 0 for the front plane and 1 for the back plane.

This matrix translates and scales objects based on a specified distance from the camera to the near clipping plane, but it doesn't consider the field of view (fov), and the z-values that it produces for objects in the distance can be nearly identical, making depth comparisons difficult. The following matrix addresses these issues, and it adjusts vertices to account for the aspect ratio of the viewport, making it a good choice for the perspective projection.

$$\text{aspect} = \frac{\text{height}}{\text{width}}$$

$$w = \text{aspect} \frac{\cos(\text{fov})}{\sin(\text{fov})}$$

$$h = \frac{\cos(\text{fov})}{\sin(\text{fov})}$$

$$q = \frac{z_{\text{far}}}{z_{\text{far}} - z_{\text{near}}}$$



In this matrix, Z_n is the z-value of the near clipping plane. The variables w , h , and Q have the following meanings. Note that fov_w and fov_h represent the viewport's horizontal and vertical fields of view, in radians.

$$w = \cot\left(\frac{fov_w}{2}\right)$$

$$h = \cot\left(\frac{fov_h}{2}\right)$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$

For your application, using field-of-view angles to define the x- and y-scaling coefficients might not be as convenient as using the viewport's horizontal and vertical dimensions (in camera space). As the math works out, the following two formulas for w and h use the viewport's dimensions, and are equivalent to the preceding formulas.

$$w = \frac{2 \cdot Z_n}{V_w}$$

$$h = \frac{2 \cdot Z_n}{V_h}$$

In these formulas, Z_n represents the position of the near clipping plane, and the V_w and V_h variables represent the width and height of the viewport, in camera space.

Lecture No.21 Triangles and Planes

Triangles

Triangles are to 3D graphics what pixels are to 2D graphics. Every PC hardware accelerator under the sun uses triangles as the fundamental drawing primitive (well ... scan line aligned trapezoids actually, but that's a hardware implementation issue). When you draw a polygon, hardware devices really draw a fan of triangles. Triangles "flesh out" a 3D object, connecting them together to form a skin or mesh that defines the boundary surface of an object. Triangles, like polygons, generally have an orientation associated with them, to help in normal calculations. The ordering of the vertices goes *clockwise* around the triangle. Figure below shows what a clockwise ordered triangle would look like.

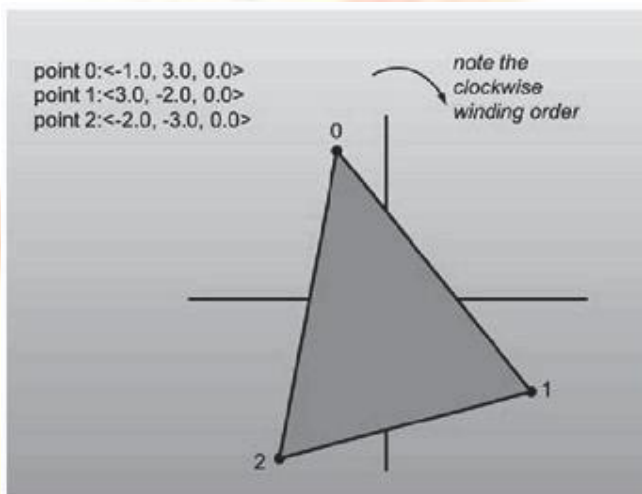


Figure: Three points in space, and the triangle connecting them

When defining a mesh of triangles that define the boundary of a solid, you set it up so that all of the triangles along the skin are ordered clockwise when viewed from the outside.

It is impossible to see triangles that face away from you. (You can find this out by computing the triangle's plane normal and performing a dot product with a vector from the camera location to a location on the plane.)

Now let's move on to the code. To help facilitate using the multiple types, I'll implement triangles structure. I only define constructors and keep the access public.

```
struct tri
{
    type v[3]; // Array access useful for loops

    tri()
    {
        // nothing
    }

    tri( type v0, type v1, type v2 )
```

```

    {
      v[0] = v0;
      v[1] = v1;
      v[2] = v2;
    }
  };

```

Strips and Fans

Lists of triangles are generally represented in one of three ways. The first is an explicit list or array of triangles, where every three elements represent a new triangle. However, there are two additional representations, designed to save bandwidth while sending triangles to dedicated hardware to draw them. They are called *triangle strips* and *triangle fans*.

Triangle fans, conceptually, look like the folding fans you see in Asian souvenir shops. They are a list of triangles that all share a common point. The first three elements indicate the first triangle. Then each new element is combined with the first element and the current last element to form a new triangle. Note that an N-sided polygon can be represented efficiently using a triangle fan

Figure below illustrates what I'm talking about.

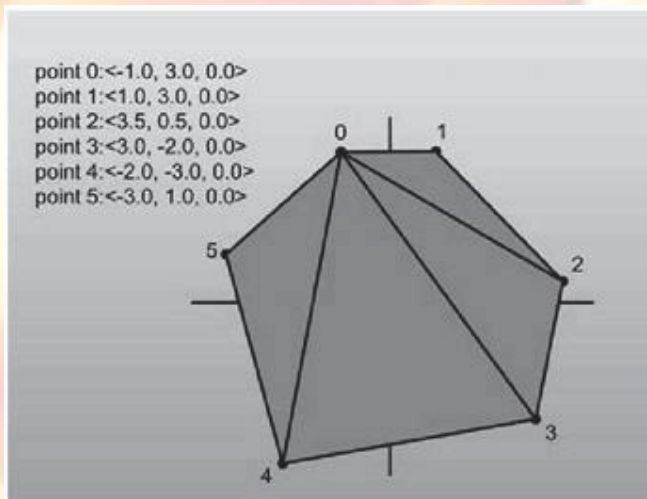


Figure: A list of points composing a triangle fan

Triangles in a triangle strip, instead of sharing a common element with all other triangles like a fan, only share elements with the triangle immediately preceding them. The first three elements define the first triangle. Then each subsequent element is combined with the two elements before it, in clockwise order, to create a new triangle. See Figure below for an explanation of strips.

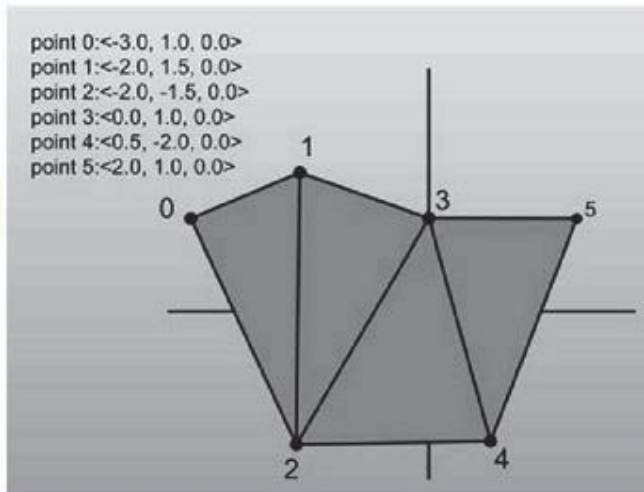


Figure: A list of points composing a triangle strip

Planes

The next primitive to discuss is the plane. Planes are to 3D what lines are in 2D; they're $n-1$ dimensional hyperplanes that can help you accomplish various tasks. Planes are defined as infinitely large, infinitely thin slices of space, like big pieces of paper. Triangles that make up your model each exist in their own plane. When you have a plane that represents a slice of 3D space, you can perform operations like classification of points and polygons and clipping.

So how do you represent planes? Well it is best to build a structure from the equation that defines a plane in 3D. The implicit equation for a plane is:

$$ax + by + cz + d = 0$$

What do these numbers represent? The triplet $\langle a, b, c \rangle$ represents what is called the normal of the plane. A *normal* is a unit vector that, conceptually speaking, sticks directly out of a plane. A stronger mathematical definition would be that the normal is a vector that is perpendicular to all of the points that lie in the plane.

The d component in the equation represents the distance from the plane to the origin. The distance is computed by tracing a line towards the plane until you hit it. Finally the triplet $\langle x, y, z \rangle$ is any point that satisfies the equation. The set of all points $\langle x, y, z \rangle$ that solve the equation is exactly all the points that lie in the plane.

All of the pictures I'm showing you will be of the top-down variety, and the 3D planes will be on edge, appearing as 2D lines. This makes figure drawing much easier.

Following are two examples of planes. The first has the normal pointing away from the origin, which causes d to be negative (try some sample values for yourself if this doesn't make sense). The second has the normal pointing towards the origin, so d is positive. Of course, if the plane goes through the origin, d is zero (the distance from the plane to the origin is zero). Figures 1 and Figure 2 provide some insight into this relation.

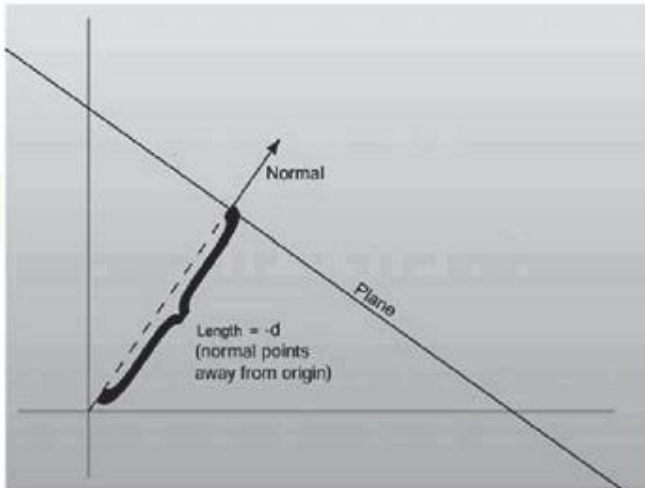


Figure 1: d is negative when the normal faces away from the origin

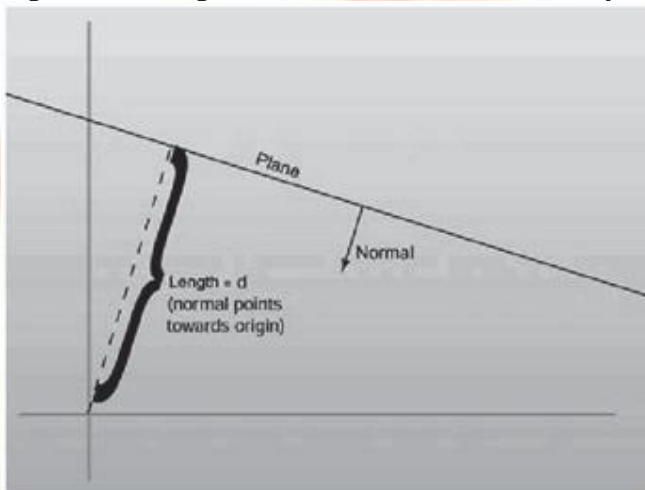


Figure 2: d is positive when it faces towards the origin

It's important to notice that technically the normal $\langle a, b, c \rangle$ does not have to be unit-length for it to have a valid plane equation. But since things end up nicer if the normal is unit-length.

Constructing a plane given three points that lie in the plane is a simple task. You just perform a cross product between the two vectors made up by the three points

$$\langle \text{point}_2 - \text{point}_0 \rangle$$

$$\langle \text{point}_1 - \text{point}_0 \rangle$$

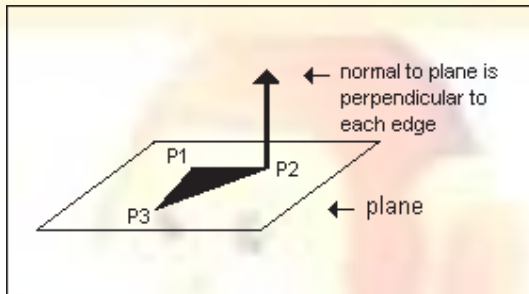
and find a normal for the plane. After generating the normal and making it unit length, finding the d value for the plane is just a matter of storing the negative dot product of the normal with any of the points. This holds because it essentially solves the plane equation above for d . Of course plugging a point in the plane equation will make it equal 0, and this constructor has three of them. Following has the code to construct a plane from three points.

To calculate a plane from 3 given points we first calculate the normal. If we imagine the 3 points form three edges in the plane then we can take two of the edges and calculate the

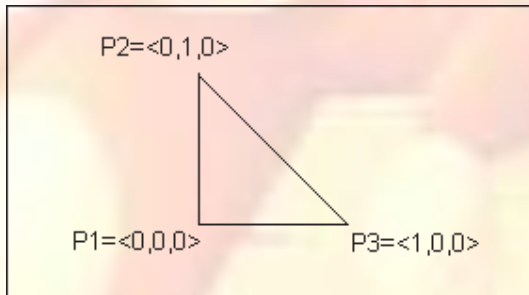
cross-product between them. The resulting directional vector will be the normal, and then we can plug any of the 3 known points into the plane equation to solve for k. For points p_1, p_2 and p_3 we get:

$$\text{normal} = (p_1 - p_2) \times (p_3 - p_2)$$

$$k = \text{normal} \cdot p_1$$



Note that it is extremely important to keep track of which direction your points are stored in. Let's take 3 points stored in clockwise direction in the x/y plane:



The normal to the plane these 3 points define is:

$$\begin{aligned} \text{normal} &= (p_1 - p_2) \times (p_3 - p_2) \\ &= (0, -1, 0) \times (1, -1, 0) \\ &= \langle (-1) \cdot 0 - 0 \cdot (-1), 0 \cdot 1 - 0 \cdot 0, 0 \cdot (-1) - (-1) \cdot 1 \rangle \\ &= \langle 0, 0, 1 \rangle \end{aligned}$$

ie the z axis. If we were to store the points counter-clockwise the normal calculated would be $\langle 0, 0, -1 \rangle$, which is still the z axis but in the "opposite" direction. It's important to keep track of these things since we often need plane equations to be correct in order to determine which side of a polygon an object (such as the view point) is on.

Constructing a plane from three points on the plane:

$$\begin{aligned} \text{Normal vector} &= n \\ n &= \text{cross product} \left((b-a), (c-a) \right) \\ \text{Normalize}(n); &\text{ find a unit vector} \end{aligned}$$

$$d = - \text{dot product}(n, a)$$

If you already have a normal and also have a point on the plane, the first step can be skipped.

Constructing a plane from a normal and a point on the plane:

Normalize(n); find a unit vector
 $d = - \text{dot product}(n,a)$

This brings up an important point. If you have an n -sided polygon, nothing discussed up to this point is forcing all of the points to be coplanar. However, problems can crop up if some of the points in the polygon aren't coplanar. For example, when I discuss back-face culling in a moment, you may misidentify what is actually behind the polygon, since there won't be a plane that clearly defines what is in front of and what is behind the plane. That is one of the advantages of using triangles to represent geometry—three points define a plane exactly.

Defining Locality with Relation to a Plane

One of the most important operations planes let you perform is defining the location of a point with respect to a plane. If you drop a point into the equation, it can be classified into three cases: in front of the plane, in back of the plane, or coplanar with the plane. Front is defined as the side of the plane the normal sticks out of.

Here, once again, precision will rear its ugly head. Instead of doing things the theoretical way, having the planes infinitely thin, I'm going to give them a certain thickness of (you guessed it) epsilon.

How do you orient a point in relation to a plane? Well, simply plug x , y , and z into the equation, and see what you get on the right side. If you get zero (or a number close enough to zero by plus or minus epsilon), then the point satisfied the equation and lies on the plane. Points like this can be called coplanar. If the number is greater than zero, then you know that you would have to travel farther along the origin following the path of the normal than you would need to go to reach the plane, so the point must be in front of the plane. If the number is negative, it must be behind the plane. Note that the first three terms of the equation simplify to the dot product of the input vector and the plane normal. Figure below has a visual representation of this operation.

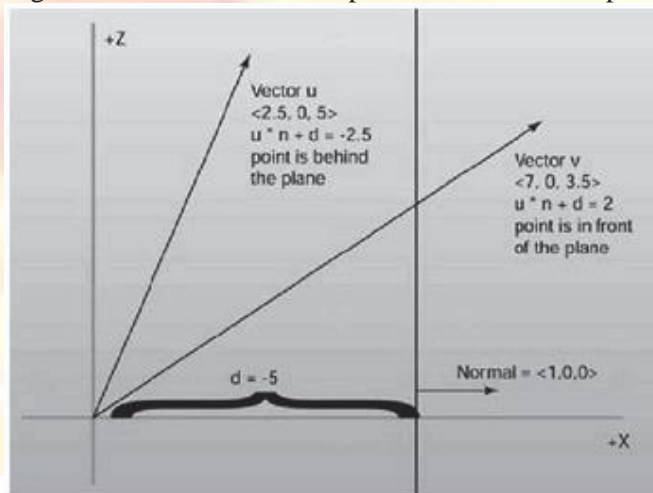


Figure: Classifying points with respect to a plane

Once you have code to classify a point, classifying other primitives, like polygons, becomes pretty trivial, as shown below. The one issue is there are now four possible definition states when the element being tested isn't infinitesimally small. The element may be entirely in front of the plane, entirely in back, or perfectly coplanar. It may also be

partially in front and partially in back. I'll refer to this state as *splitting* the plane. It's just a term; the element isn't actually splitting anything.

Back-face Culling

Now that you know how to define a point with respect to a plane, you can perform back-face culling, one of the most fundamental optimization techniques of 3D graphics.

Let's suppose you have a triangle whose elements are ordered in such a fashion that when viewing the triangle from the front, the elements appear in clockwise order. Back-face culling allows you to take triangles defined with this method and use the plane equation to discard triangles that are facing away. Conceptually, any closed mesh, a cube for example, will have some triangles facing you and some facing away. You know for a fact that you'll never be able to see a polygon that faces away from you; they are always hidden by triangles facing towards you. This, of course, doesn't hold if you're allowed to view the cube from its inside, but this shouldn't be allowed to happen if you want to really optimize your engine.

Rather than perform the work necessary to draw all of the triangles on the screen, you can use the plane equation to find out if a triangle is facing towards the camera, and discard it if it is not. How is this achieved? Given the three points of the triangle, you can define a plane that the triangle sits in. Since you know the elements of the triangle are listed in clockwise order, you also know that if you pass the elements in order to the plane constructor, the normal to the plane will be on the front side of the triangle. If you then think of the location of the camera as a point, all you need to do is perform a point-plane test. If the point of the camera is in front of the plane, then the triangle is visible and should be drawn.

There's an optimization to be had. Since you know three points that lie in the plane (the three points of the triangle) you only need to hold onto the normal of the plane, not the entire plane equation. To perform the back-face cull, just subtract one of the triangle's points from the camera location and perform a dot product with the resultant vector and the normal. If the result of the dot product is greater than zero, then the view point was in front of the triangle. Figure below can help explain the point.

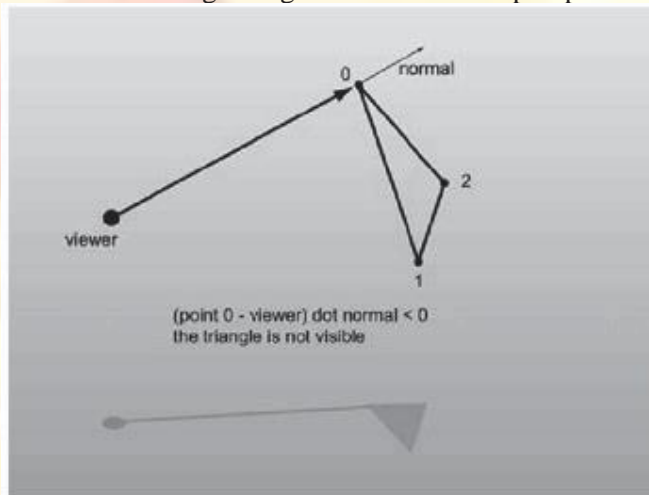


Figure: A visual example of back-face culling

In practice, 3D accelerators can actually perform back-face culling by themselves, so as the triangle rates of cards increase, the amount of manual back-face culling that is performed has steadily decreased. However, the information is useful for custom 3D engines that don't plan on using the facilities of direct hardware acceleration.

Intersection between a Line and a Plane

This occurs at the point which satisfies both the line and the plane equations.

$$\text{Line equation: } \mathbf{p} = \text{org} + \mathbf{u} * \text{dir} \quad (1)$$

$$\text{Plane equation: } \mathbf{p} * \text{normal} - \mathbf{k} = 0. \quad (2)$$

Substituting (1) into (2) and rearranging we get:

$$(\text{org} + \mathbf{u} * \text{dir}) * \text{normal} - \mathbf{k} = 0$$

$$\text{ie } \mathbf{u} * \text{dir} * \text{normal} = \mathbf{k} - \text{org} * \text{normal}$$

$$\text{ie } \mathbf{u} = (\mathbf{k} - \text{org} * \text{normal}) / (\text{dir} * \text{normal})$$

If $(\text{dir} * \text{normal}) = 0$ then the line runs parallel to the plane and no intersection occurs. The exact point at which intersection does occur can be found by plugging \mathbf{u} back into the line equation in (1).

Clipping Lines

One thing that you'll need is the ability to take two points (**a** and **b**) that are on different sides of a plane defining a line segment, and find the point making the intersection of the line with the plane.

This is easy enough to do. Think of this parametrically. Point **a** can be thought of as the point at time 0 and point **b** as the point at time 1, and the point of intersection you want to find is somewhere between those two.

Take the dot product of **a** and **b**. Using them and the inverse of the plane's *d* parameter, you can find the scale value (which is a value between 0 and 1 that defines the parametric location of the particle when it intersects the plane). Armed with that, you just use the scale value, plugging it into the linear parametric equation to find the intersection location. Figure 5.17 shows this happening visually, and Listing 5.20 has the code.

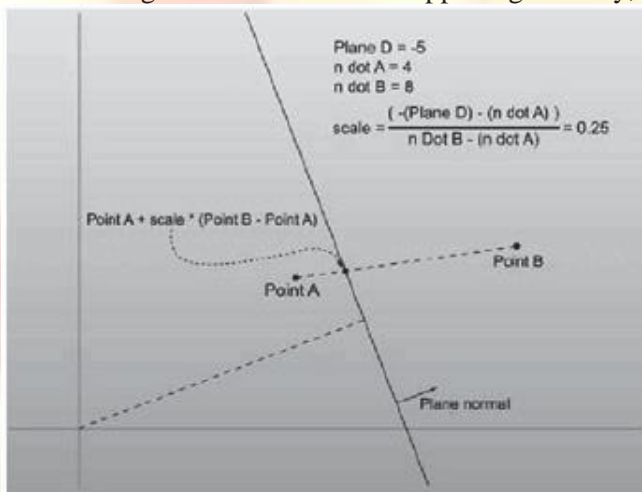


Figure 5.17: Finding the intersection of a plane and a line

Listing 5.20: plane3::Split

```
inline const point3 plane3::Split(const point3 &a, const point3 &b) const
{
    float aDot = (a * n);
    float bDot = (b * n);

    float scale = (-d - aDot) / (bDot - aDot);
    return a + (scale * (b - a));
}
```

Lecture No.22 Triangle Rasterization

Introduction

High performance triangle rasterization is a very important topic in Computer Graphics in today's world.

Triangles are the foundation of modern real time graphics, and are by far the most popular rendering primitive. Most computer games released in the last few years are almost completely dependent on triangle rasterization performance. Recently the focus of graphics performance optimization is beginning to shift to bandwidth requirements as well as transformation and lighting. Nevertheless, rasterization performance is still a factor, and this lecture will provide most of the basics of high performance triangle rasterization. Also, it will go into detail about two often neglected rendering quality improvements, sub-pixel and sub-texel accuracy. Also, smooth shading and texture mapping techniques will be described.

Solid Fill Triangle Rendering

The first step in triangle rasterization is to be able to render a solid filled triangle. All triangle drawing routines should fill the same pixels on the screen so it makes sense to start with the simplest example and work up. The goal is to draw a filled triangle by plotting pixels on the screen given three vertex points.

The first step is to sort the triangle vertices by y . Label the top vertex (x_0, y_0) , the middle vertex (x_1, y_1) , and the bottom vertex (x_2, y_2) . Now the triangle fill can be thought of as two separate routines, filling the top half (the region between y_0 and y_1) and filling the bottom half (the region between y_1 and y_2). Each of the fill routines consists of filling the triangle region one scanline at a time, using the DDA algorithm to find the x values of the beginning and the end of each pixel span to draw. The top half will use DDA to find the x values on edge₀₁ and edge₀₂. The bottom half uses DDA to find the x values on edge₁₂ and edge₀₂.

Sub-pixel Accuracy

The afore mentioned rasterization technique works well when vertex coordinates are integers, but there are some subtle changes that should be made to the DDA algorithm when the vertex coordinates do not fall on integer bounds. In essence, sub-pixel accuracy is a way of accounting for the fractional components of the vertex positions in the triangle rasterizer. The changes that need to be made are mainly used to prevent jumpiness when there are amounts of motion that are smaller than a pixel. The edges of the triangle reflect the fractional change, and without sub-pixel accuracy, the entire triangle would jump down a pixel at a time. Also, the calculations performed for sub-pixel accuracy allow for quicker edge anti-aliasing.

The idea of sub-pixel accuracy is to pre-step the x coordinate of each of the edge DDAs an amount corresponding to the fractional component of the y position of the vertex. For

notation sake we denote the upper vertex of an edge as x_a, y_a and the lower vertex as x_b, y_b . For each edge the starting x coordinate for the DDA algorithm x_s is obtained by a prestep amount $x_{prestep}$ to the original x coordinate of the vertex:

The pre step amount ' x_f ' for x , is calculated by multiplying $\partial x/\partial y$ by the fix up distance, y_f . This fix up distance is just the distance of y_a to the next lowest scan line. For clarification purposes, here is some pseudo code for the sub-pixel accurate DDA, which can be used to find endpoints for the pixel spans in our triangle rasterizer. This technique can be used to draw sub-pixel accurate lines also.

```
SubPixDDA( float xa, float ya, float xb, float yb)
{
  int yai,ybi, /*scanline range to draw*/
      y;      /*iterator for scanline*/

  float xp, /*prestep in x */
        x, /*current x position*/
        yf, /*fractional distance in y */
        dxdy; /*amount to change x by per scanline (1/slope) */

  dxdy=(xb-xa)/(yb-ya); /*perform slope calculation using true */
  /*vertex positions*/
  yai=ceiling(ya);
  ybi=ceiling(yb);

  yf=yai-ya;
  xp=(dxdy*yf);
  x=xa+xp;

  for(y=yai;y<ybi;y++) /*iterate over scanlines*/
  {
    /*do scanline stuff using x,y */
    x+=dxdy;
  }
}
```

Smooth Shaded Triangle Rasterization

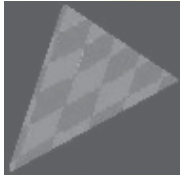
So far we have described a method for solid filled triangle rasterization, but there are a variety of other fill types that are use. Smooth shaded triangles can be used to approximate the effects of lighting over a surface. They can be used for light falloff, or can be used to give the appearance to a curved surface.

The idea behind smooth shading is to linearly interpolate the vertex colors over the triangle being drawn. Luckily for us, we already have the tool to do this, DDA. In fact, drawing smooth shaded polygons is not much more difficult than drawing solid filled ones. The vertex colors must be interpolated along each edge of the triangle using DDA. This gives us a separate pair of colors for the beginning and end of each pixel span for

each scan line. The last step that needs to be performed is to use DDA to interpolate the colors across each pixel span.

To do smooth shading with RGB color, you must use separate DDA interpolation routines for the red, green, and blue components of the color. Also note that inside the smooth shading routine r , g , and b must be represented as a type with a fractional component (type float is a good choice). To avoid visual artifacts it is recommended that you use the sub-texel accuracy technique that is described further on in these notes.

Texture Mapped Triangle Rasterization



Another common triangle fill method is called texture mapping. Texture mapping is a technique for interpolating an image over the triangle being rasterized. The image being interpolated is known as a texture map, and each pixel in the texture map is known as a texel. Because it allows for the use of images to represent a surface on an object, it has the potential to greatly reduce the number of triangles needed to represent an object. In addition to this, texture mapping can also be used to simulate the effects of complex lighting conditions on an object.

This section describes bilinear texture mapping, which is the simplest technique to implement. In order to perform bilinear texture mapping, each vertex contains a u , v texture coordinate. This specifies the location in the texture map that this vertex corresponds to. Given these texture coordinates, texture mapping isn't much more difficult than smooth shading. The texture coordinates u , v are interpolated over the triangle using DDA just like the r , g , and b values are in smooth shading. The difference is that the resulting u , v location for every pixel is used to lookup a color value in the texture map image for the pixel to be drawn.

However, there is the problem how to deal with the fractional component of the u , v values; looking up color values in the texture map requires integer coordinates.

One technique is to round the u , v values to the nearest integer. This is the quickest approach, but produces a blocky looking triangle image when the texture map is small in comparison to the triangle size. Most software based texture mapping routines used in computer games take use this approach because of the speed advantages. However, the majority of hardware based texture mapping routines has an option to do bilinear sampling. Bilinear sampling uses the fractional component of the u , v coordinate to perform a weighted average of 4 adjacent texel colors. The fractional components of u , v are used to find the distance of u , v from the texels themselves. This distance is used as the weighting, and the formula for the pixel color to be drawn C_{bilin_samp} is:

Here is some pseudocode to do this:

```
color BilinearSampling(float u, float v, color texMap[256][256])
```

```
{
    color c00,c01,c10,c11;
    int u0,u1,v0,v1;
    float ufrac,vfrac;

    u0=floor(u);
    u1=ceiling(u);
    v0=floor(v);
    v1=ceiling(v);
    ufrac=u-u0;
    vfrac=v-v0;

    c00=texMap[u0][v0];
    c01=texMap[u0][v1];
    c10=texMap[u1][v0];
    c11=texMap[u1][v1];

    return( vfrac * (ufrac*c11 + (1-ufrac)*c01) +
           (1-vfrac) * (ufrac*c10 + (1-ufrac)*c00) );
}
```

Sub-Textel Accuracy

The disparity between integer screen pixel locations and the mathematical equations for the triangle also causes problems for texture mapping, and smooth shading. Any value that is interpolated over the triangle such as r, g, and b for smooth shading and u and v for texture mapping must take into account the fractional component of the vertex information. Taking these fractional quantities into account is called sub-textel accuracy, for the reason that it is used most commonly with texture mapping. In actuality, sub-textel accuracy can be applied to any quantity interpolated over the triangle. Without sub-textel accuracy, the texture will visibly jump around by a pixel when the triangle undergoes small amounts of motion.

The sub-textel accurate DDA interpolators for texture mapping are very similar to the sub-pixel accurate DDA routine presented earlier. For each edge of the triangle, the sub-textel DDA for the interpolated values is identical to the sub-pixel DDA, when u or v is substituted for x . However, for each scan line, the beginning and end x locations of the pixel span have fractional components which need to be accounted for. To interpolate the texels coordinates correctly over the pixel span for each scan line, a subtextel accurate pixel span DDA is required. Luckily for us, this formulation is also virtually identical to the sub-pixel DDA. All that need to be done is to substitute u or v for x in the original, and substitute x for y in the original.

Flat Filling Triangles



Drawing triangle (or in general convex polygon, but as we discussed we will use only triangles) is very simple. The basic idea of the line triangle drawing algorithm is as follows.

For each scan line (horizontal line on the screen), find the points of intersection with the edges of the triangle. Then, draw a horizontal line between intersections and do this for all scan lines.

But how can we find these points quickly?
Using linear interpolation!

We have 3 vertices and we want to find coordinates of all points belonging to segments determined by these vertices.

Assume we have segment given by points:

(x_a, y_a) and (x_b, y_b) .

Our task is to find points: (x_c, y_a+1) , (x_d, y_a+2) , ..., (x_m, y_b-1) , (x_n, y_b) .

Notice that x_a changes to x_b in $(y_b - y_a)$ steps.

We also have:

$$x_a = x_a + 0 * (x_b - x_a) / (y_b - y_a),$$

$$x_b = x_a + (y_b - y_a) * (x_b - x_a) / (y_b - y_a)$$

$$\text{and, in general, } x_i = x_a + (y_i - y_a) * \text{delta},$$

where $\text{delta} = (x_b - x_a) / (y_b - y_a)$.

The general function for linear interpolation is:

$$f(X) = A + X * ((B - A) / \text{steps}) \text{ where we slide from } A \text{ to } B \text{ in } \text{steps} \text{ steps}$$

Here is pseudo code for a triangle filling algorithm.

The coordinates of vertices are $(A. x, A. y)$, $(B. x, B. y)$, $(C. x, C. y)$; we assume that $A. y \leq B. y \leq C. y$ (you should sort them first)

dx_1, dx_2, dx_3 are deltas used in interpolation

Horizontal line draws horizontal segment with coordinates
 $(S. x, Y)$, $(E. x, Y)$

$S. x, E. x$ are left and right x-coordinates of the segment we have to draw

$S = A$ means that $S. x = A. x$; $S. y = A. y$;

```
if (B.y - A.y > 0) dx1 = (B.x-A.x) / (B.y-A.y)
else dx1=0;
```

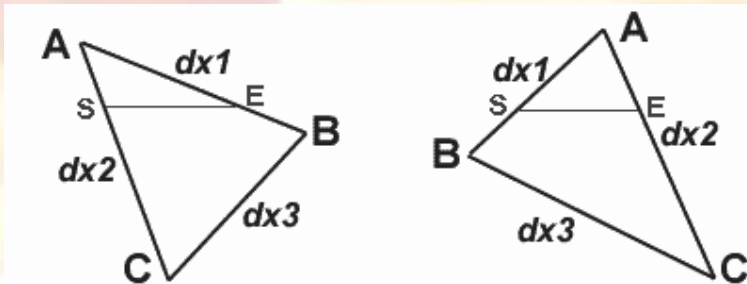
```
if (C.y - A.y > 0) dx2=(C.x-A.x)/(C.y-A.y)
else dx2=0;
```

```
if (C.y-B.y > 0) dx3=(C.x-B.x)/(C.y-B.y)
else dx3=0;
```

```
S = E = A
```

```
if(dx1 > dx2)
{
    for(; S.y<=B.y; S.y++, E.y++, S.x+=dx2, E.x += dx1)
        horizontal line (S.x,E.x,S.y, E.x,color);
    E=B;
    for(;S.y<=C.y;S.y++,E.y++,S.x+=dx2,E.x+=dx3)
        horizontal line(S.x,E.x,S.y, E.x,color);
}
else
{
    for(;S.y<=B.y;S.y++,E.y++,S.x+=dx1,E.x+=dx2)
        horizontal line(S.x,E.x,S.y, E.x,color);
    S=B;
    for(;S.y<=C.y;S.y++,E.y++,S.x+=dx3,E.x+=dx2)
        horizontal line(S.x,E.x,S.y, E.x,color);
}
```

I ought to explain what is the comparison $dx1 > dx2$ for. It's optimization trick: in the horizontal line routine, we don't need to compare the x's (S.x is always less than or equal to E.x).



Gouraud Shading



The idea of gouraud and flat triangle is nearly the same. Gouraud takes only three parameters more (the color value of each of the vertices), and the routine just interpolates among them drawing a beautiful, shaded triangle.

You can use 256-colors mode, in which vertices' colors are simply indices to palette or hi-color mode (recommended).

Flat triangle interpolated only one value (x in connection with y), 256 colors gouraud needs three (x related to y, color related to y, and color related to x), hi-color gouraud needs seven (x related to y, red, green and blue components of color related to y, and color related to x (also three components))

Drawing a gouraud triangle, we add only two parts to the flat triangle routine. The horizline routine gets a bit more complicated due to the interpolation of the color value related to x but the main routine itself remains nearly the same.

We'll give you a full gouraud routine because good pseudo code is better than the best description:

the coordinates of vertices are $(A.x, A.y)$, $(B.x, B.y)$, $(C.x, C.y)$ we assume that $A.y \leq B.y \leq C.y$ (you should sort them first)

vertex A has color $(A.r, A.g, A.b)$, B $(B.r, B.g, B.b)$, C $(C.r, C.g, C.b)$, where $X.r$ is color's red component, $X.g$ is color's green component and $X.b$ is color's blue component

$dx1, dx2, dx3$ are deltas used in interpolation of x-coordinate

$dr1, dr2, dr3$, $dg1, dg2, dg3$, $db1, db2, db3$ are deltas used in interpolation of color's components

putpixel(P) plots a pixel with coordinates $(P.x, P.y)$ and color $(P.r, P.g, P.b)$

$S=A$ means that $S.x=A.x$; $S.y=A.y$; $S.r=A.r$; $S.g=A.g$; $S.b=A.b$;

Drawing triangle:

```
if (B.y-A.y > 0) {
    dx1=(B.x-A.x)/(B.y-A.y);
    dr1=(B.r-A.r)/(B.y-A.y);
    dg1=(B.g-A.g)/(B.y-A.y);
    db1=(B.b-A.b)/(B.y-A.y);
} else
    dx1=dr1=dg1=db1=0;
```

```
if (C.y-A.y > 0) {
    dx2=(C.x-A.x)/(C.y-A.y);
    dr2=(C.r-A.r)/(C.y-A.y);
    dg2=(C.g-A.g)/(C.y-A.y);
```

```

        db2=(C.b-A.b)/(C.y-A.y);
    } else
        dx2=dr2=dg2=db2=0;

    if (C.y-B.y > 0) {
        dx3=(C.x-B.x)/(C.y-B.y);
        dr3=(C.r-B.r)/(C.y-B.y);
        dg3=(C.g-B.g)/(C.y-B.y);
        db3=(C.b-B.b)/(C.y-B.y);
    } else
        dx3=dr3=dg3=db3=0;

    S=E=A;
    if(dx1 > dx2) {
        for(;S.y<=B.y;S.y++,E.y++) {
            if(E.x-S.x > 0) {
                dr=(E.r-S.r)/(E.x-S.x);
                dg=(E.g-S.g)/(E.x-S.x);
                db=(E.b-S.b)/(E.x-S.x);
            } else
                dr=dg=db=0;
            P=S;
            for(;P.x < E.x;P.x++) {
                putpixel(P);
                P.r+=dr; P.g+=dg; P.b+=db;
            }
            S.x+=dx2; S.r+=dr2; S.g+=dg2; S.b+=db2;
            E.x+=dx1; E.r+=dr1; E.g+=dg1; E.b+=db1;
        }

        E=B;
        for(;S.y<=C.y;S.y++,E.y++) {
            if(E.x-S.x > 0) {
                dr=(E.r-S.r)/(E.x-S.x);
                dg=(E.g-S.g)/(E.x-S.x);
                db=(E.b-S.b)/(E.x-S.x);
            } else
                dr=dg=db=0;
            P=S;
            for(;P.x < E.x;P.x++) {
                putpixel(P);
                P.r+=dr; P.g+=dg; P.b+=db;
            }
            S.x+=dx2; S.r+=dr2; S.g+=dg2; S.b+=db2;
            E.x+=dx3; E.r+=dr3; E.g+=dg3; E.b+=db3;
        }
    } else {
        for(;S.y<=B.y;S.y++,E.y++) {
            if(E.x-S.x > 0) {
                dr=(E.r-S.r)/(E.x-S.x);

```

```

        dg=(E.g-S.g)/(E.x-S.x);
        db=(E.b-S.b)/(E.x-S.x);
    } else
        dr=dg=db=0;

    P=S;
    for(;P.x < E.x;P.x++) {
        putpixel(P);
        P.r+=dr; P.g+=dg; P.b+=db;
    }
    S.x+=dx1; S.r+=dr1; S.g+=dg1; S.b+=db1;
    E.x+=dx2; E.r+=dr2; E.g+=dg2; E.b+=db2;
}

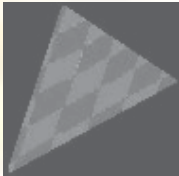
S=B;
for(;S.y<=C.y;S.y++,E.y++) {
    if(E.x-S.x > 0) {
        dr=(E.r-S.r)/(E.x-S.x);
        dg=(E.g-S.g)/(E.x-S.x);
        db=(E.b-S.b)/(E.x-S.x);
    } else
        dr=dg=db=0;

    P=S;
    for(;P.x < E.x;P.x++) {
        putpixel(P);
        P.r+=dr; P.g+=dg; P.b+=db;
    }
    S.x+=dx3; S.r+=dr3; S.g+=dg3; S.b+=db3;
    E.x+=dx2; E.r+=dr2; E.g+=dg2; E.b+=db2;
}
}

```

I hope you are familiar with idea of interpolation now.

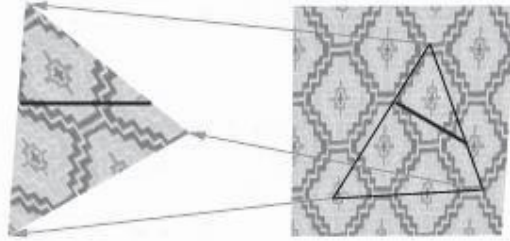
Textured Triangles



We can also apply any bitmap on triangle for filling it.

I'll show you the idea of linear (or 'classical') texture mapping (without perspective correction). Linear mapping works pretty well (read: fast) in some scenes, but perspective correction is in some way needed in most 3D systems.

Again we're using the idea of interpolation: now we'll code a texture triangle filler. And again the idea is perfectly the same, only two more values to interpolate, that is five values total. In texture mapping, we interpolate x , u , and v related to y , and u and v related to x (u and v are coordinates in the 2D bitmap space). The situation is maybe easier to understand by looking at the following picture:



The left triangle is the triangle which is drawn onto the screen. There's a single scanline (one call to the horzline routine) pointed out as an example. The triangle on the right is the same triangle in the bitmap space, and there's the same scanline drawn from another point of view into it, too. So we need just to interpolate, interpolate, and once more interpolate in texture filler - an easy job if you've understood the idea of gouraud filler. An optimization trick: the color deltas in gouraud and (u,v) coordinate deltas in texture remain constant, so we need to calculate them only once per polygon. Let's take the u delta in linear texturing as an example. Assume, that $dx2 <= dx3$ (we are using the same symbols like in flat and gouraud filler). As we know, we need to interpolate $S.u$ to $E.u$ in the horzline routine in $(S.x - E.x)$ steps. We are in the need of a u delta (du) which would be the same for the whole polygon. So instead of calculating in each scanline this:

$$du = (E.u - S.u) / (E.x - S.x),$$

we do like this in the setup part of the polygon routine: We know that

$$S.x = A.x + (B.y - A.y) * dx1,$$

$$S.u = A.u + (B.y - A.y) * du1,$$

$$E.x = B.x = A.x + (B.y - A.y) * dx2,$$

$$E.u = B.u = A.u + (B.y - A.y) * du2,$$

When

$y = B.y$ (when y is the y -coordinate of the second vertex).

When we place the values of the variables $S.u, E.u, S.x$ and $E.x$ (above) to the u delta statement,

$$du = (E.u - S.u) / (E.x - S.x),$$

we get the following statement as a result:

$$du = \frac{[A.u + (B.y - A.y) * du2] - [A.u + (B.y - A.y) * du1]}{[A.x + (B.y - A.y) * dx2] - [A.x + (B.y - A.y) * dx1]}$$

$$du = \frac{(B.y - A.y) * (A.u - A.u + du2 - du1)}{(B.y - A.y) * (A.x - A.x + dx2 - dx1)}$$

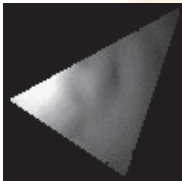
$$du = \frac{du2 - du1}{dx2 - dx1}$$

$$\text{innerUdelta} = \frac{\text{outerUdelta2} - \text{outerUdelta1}}{\text{outerXdelta2} - \text{outerXdelta1}}$$

Nice! But what if $dx2 = dx1$? This of course means that the polygon is just one line, so du doesn't need any specific value; zero does the job very well.

Note! I find it hard to get good results using fixed point math because of inadequate precision.

Environmental Mapping



As I said in 'shading' part, the way demos do environment mapping is very simple. Take the X and Y components of your pseudo-normal vectors (perpendicular to vertices), and use them to index your texture map!

Your formulae would be:

$$U = N.x * 128 + 127$$

$$V = N.y * 128 + 127 \text{ (assuming 256x256 texture maps).}$$

Or in general

$$U = N.x * (\text{width} / 2) + (\text{width} / 2) - 1$$

$$V = N.y * (\text{height} / 2) + (\text{height} / 2) - 1$$



Using texturing and shading at the same time is quite straightforward to implement: the basic idea being that we just interpolate the values of both texture and shade and blend them in a suitable ratio (alpha-blending).

Lecture No.23 Lighting I

MATHEMATICS OF COLOR IN COMPUTER GRAPHICS

It is important to understand how color is represented in computer graphics so that we can manipulate it effectively. A color is usually represented in the graphics pipeline by a three-element vector representing the intensities of the red, green, and blue components, or for a more complex object, by a four-element vector containing an additional value called the *alpha* component that represents the opacity of the color. Thus we can talk about **rgb** or **rgba** colors and mean a color that's made up of either three or four elements. There are many different ways of representing the intensity of a particular color element. *Colors can also be represented as floating point values in the range [0,1].*

Nowadays every PC we can buy has hardware that can render images with thousands or millions of individual colors. Rather than have an array with thousands of color entries, the images instead contain explicit color values for each pixel. A 16-bit display is named since each pixel in a 16-bit image is taken up by 16 bits (2 bytes): 5 bits of red information, 6 bits of green information, and 5 bits of blue information. Incidentally, the extra bit (and therefore twice as much color resolution) is given to green because our eyes are more sensitive to green. A 24-bit display, of course, uses 24 bits, or 3 bytes per pixel, for color information. This gives 1 byte, or 256 distinct values each, for red, green, and blue. This is generally called *true color*, because 256^3 (16.7 million) colors is about as much as your eyes can discern, so more color resolution really isn't necessary, at least for computer monitors.

Finally, there is 32-bit color, something seen on most new graphics cards. Many 3D accelerators keep 8 extra bits per pixel around to store transparency information, which is generally referred to as the *alpha channel*, and therefore take up 4 bytes, or 32 bits, of storage per pixel. Rather than re-implementing the display logic on 2D displays that don't need alpha information, these 8 bits are usually just wasted.

Representing Color

Before we can go about giving color to anything in a scene, we need to know how to represent color! Usually we use the same red, green, and blue (rgb) channels discussed above, but for this there will also be a fourth component called *alpha*. The alpha component stores transparency information about a surface. Practically we will use two structures to ease the color duties: `color3` and `color4`. They both use floating-point values for their components; `color3` has red, green, and blue, while `color4` has the additional fourth component of alpha in there.

Colors aren't like points—they have a fixed range. Each component can be anywhere between 0.0 and 1.0 (zero contribution of the channel or complete contribution). If performing operations on colors, such as adding them together, the components may rise above 1.0 or below 0.0.

The code for color4 appears below.

The color4 structure

```

struct color4
{
    union {
        struct
        {
            float r, g, b, a; // Red, Green, and Blue color data
        };
        float c[4];
    };
};

color4(){}

color4( float inR, float inG, float inB, float inA ):
    r( inR ), g( inG ), b( inB ), a( inA )
{
}

color4( const color3& in, float alpha = 1.f )
{
    r = in.r;
    g = in.g;
    b = in.b;
    a = alpha;
}

color4( unsigned long color )
{
    b = (float)(color&255) / 255.f;
    color >>= 8;
    g = (float)(color&255) / 255.f;
    color >>= 8;
    r = (float)(color&255) / 255.f;
    color >>= 8;
    a = (float)(color&255) / 255.f;
}

unsigned long MakeDWord()
{
    unsigned long iA = (int)(a * 255.f) << 24;
    unsigned long iR = (int)(r * 255.f) << 16;
    unsigned long iG = (int)(g * 255.f) << 8;
    unsigned long iB = (int)(b * 255.f);
    return iA | iR | iG | iB;
}

// if any of the values are >1, cap them.

```

```
void Saturation()  
  
  {  
    if( r > 1 )  
      r = 1.f;  
    if( g > 1 )  
      g = 1.f;  
    if( b > 1 )  
      b = 1.f;  
    if( a > 1 )  
      a = 1.f;  
    if( r < 0.f )  
      r = 0.f;  
    if( g < 0.f )  
      g = 0.f;  
    if( b < 0.f )  
      b = 0.f;  
    if( a < 0.f )  
      a = 0.f;  
  }  
};
```

We should also point out that when dealing with colors, particularly with some of the subtleties that we'll be getting into with lights and shades, we should understand the gamut of the target device. This is where our beautiful clean mathematics meets the real world. The gamut of a device is simply the physical range of colors the device can display. Typically, a high-quality display has a better gamut than a cheap one. A good printer has a gamut that is significantly different from that of a monitor. If we are interested in getting some color images for printing, we shall have to do some manipulation on the color values to make the printed image look like the one our program generated on the screen. We should also be aware that there are color spaces other than the RGB color space. HSV (hue, saturation, and value) is one that's typically used by printers, for example.

WHY WE MIGHT WANT 128-BIT COLOR

In one of the early magazines articles of *Mike Abrash* [ABRASH 1992], he tells a story about going from a 256-color palette to hardware that supported 256 levels for each RGB color—16 million colors! What would we do with all those colors? He goes on to tell of a story by Sheldon Linker at the eighth Annual Computer Graphics Show on how the folks at the Jet Propulsion Lab back in the 1970s had a printer that could print over 50 million distinct colors. As a test, they printed out words on paper where the background color was only one color index from the word's color. To their surprise, it was easy to discern the words—the human eye is very sensitive to color graduations and edge detection. The JPL team then did the same tests on color monitors and discovered that only about 16 million colors could be distinguished. It seems that the eye is (not too surprisingly) better at perceiving detail from reflected light (such as from a printed page) than from emissive light (such as from a CRT). The moral is that the eye is a lot more perceptive than you

might think. Twenty four-bits of color really is not that much range, particularly if we are performing multiple passes. Round-off error can and will show up if we aren't careful!

An example of the various gamuts is shown in the figure below. The CIE diagrams are the traditional way of displaying perceived color space, which, we should note, is very different from the linear color space used by today's graphics hardware. The colored area is the gamut of the human eye. The gamut of printers and monitors are subsets of this gamut.

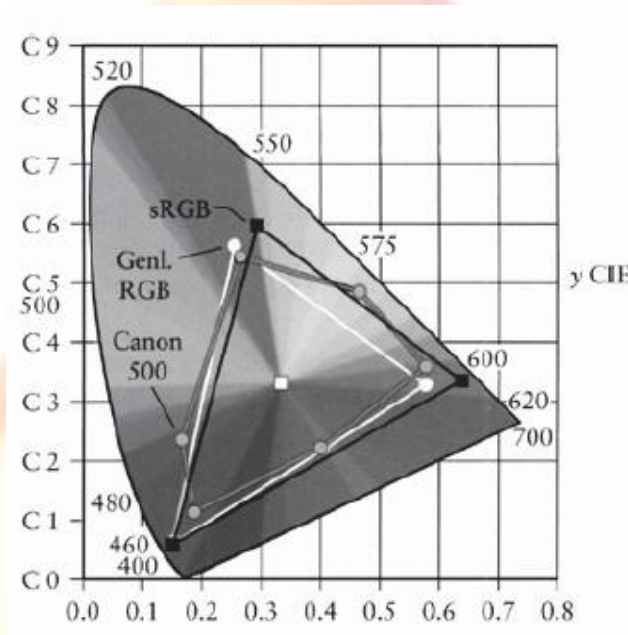


Figure 1: The 1931 CIE diagram shows the gamut of the eye and the lesser gamut of output devices.

Multiplying Color Values

First we need to be aware of how to treat colors. The calculation of the color of a particular pixel depends, for example, on the surface's material properties that we've programmed in, the color of the ambient light (*lighting model*), the color of any light shining on the surface (perhaps of the angle of the light to the surface), the angle of the surface to the viewpoint, the color of any fog or other scattering material that's between the surface and the viewpoint, etc. No matter how you are calculating the color of the pixel, it all comes down to color calculations, at least on current hardware, on **rgb** or **rgba** vectors where the individual color elements are limited to the [0,1] range. Operations on colors are done piecewise—that is, even though we represent colors as **rgb** vectors, they aren't really vectors in the mathematical sense. Vector multiplication is different from the operation we perform to multiply color. We'll use the \otimes symbol to indicate such piecewise multiplication.

Colors are multiplied to describe the interaction between a surface and a light source. The colors of each are multiplied together to estimate the reflected light color—this is the color of the light that this particular light reflects off this surface. The problem with the

standard **rgb** model is just that we're simulating the entire visible spectrum by three colors with a limited range.

Let's start with a simple example of using reflected colors. Later on we will discuss on lighting, we'll discover how to calculate the intensity of a light source, but for now, just assume that we've calculated the intensity of a light, and it's a value called i_d . This intensity of our light is represented by, say, a nice lime green color.

Thus

$$\text{light color } i_d = [0.34765, 0.92578, 0.24609]$$

Let's say we shine this light on a nice magenta surface given by c_s .

$$\text{surface color } c_s = [0.86719, 0.00000, 0.98828]$$

So, to calculate the color contribution of this surface from this particular light, we perform a piecewise multiplication of the color values.

$$\begin{aligned} i_d \otimes c_s &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\ &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\ &= [0.30148, 0.00000, 0.243210] \end{aligned}$$

Note: Piecewise multiplication is denoted by $i_d \otimes c_s$ that is element-by-element multiplication, used in color operations, where the vector just represents a convenient notation for an array of scalars that are operated on simultaneously but independently.

This gives us the dark plum color shown in figure below. We should note that since the surface has no green component, that no matter what value we used for the light color, there would *never* be any green component from the resulting calculation. Thus a pure green light would provide no contribution to the intensity of a surface if that surface contained a zero value for its green intensity. Thus it's possible to illuminate a surface with a bright light and get little or no illumination from that light. We should also note that using anything other than a full-bright white light $[1,1,1]$ will involve multiplication of values less than one, which means that using a single light source will only illuminate a surface to a maximum intensity of its color value, never more. This same problem also happens when a texture is modulated by a surface color. The color of the surface will be multiplied by the colors in the texture. If the surface color is anything other than full white, the texture will become darker. Multiple texture passes can make a surface very dark very quickly.



Figure 2: Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

Given that using a colored light in a scene makes the scene darker, how do we make the scene brighter? There are a few ways of doing this. Given that color multiplication will never result in a brighter color, it's offset a bit since we end up summing all the light contributions together, which, as we'll see in the next section, brings with it its own problems. But if we are just interested in increasing the brightness on one particular light or texture, one way is to use the API (*Library routines e.g. OpenGL or DirectX*) to artificially brighten the source—this is typically done with texture preprocessing. Or, we can artificially brighten the source, be it a light or a texture, by adjusting the values after we modulate them.

Dealing with Saturated Colors

On the other hand, what if we have *too* much contribution to a color? While the colors of lights are modulated by the color of the surface, *each* light source that illuminates the surface is added to the final color. All these colors are summed up to calculate the final color. Let's look at such a problem. We'll start with summing the reflected colors off a surface from two lights. The first light is an orange color and has *rgb* values [1.0,0.49,0.0], and the second light is a nice light green with *rgb* values [0.0,1.0,0.49]. Summing these two colors yields [1.0, 1.49, 0.49], which we can't display because of the values larger than one figure below shows.

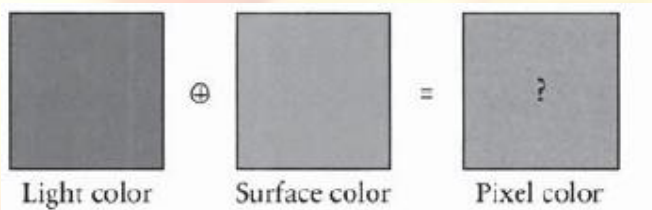


Figure 3: Adding colors can result in colors that are outside the displayable range.

So, what can be done when color values exceed the range that the hardware can display? It turns out that there are three common approaches [HALL 1990].

Clamping the color values is implemented in hardware, so for shaders (*technology used in today computer graphics for lighting and shading*), it's the default, and it just means that we clamp any values outside the [0,1] range. Unfortunately, this results in a shift in the color.

The second most common approach is to scale the colors by the largest component. This maintains the color but reduces the overall intensity of the color.

The third is to try to maintain the intensity of the color by shifting (or clipping) the color toward pure bright white by reducing the colors that are too bright while increasing the other colors and maintaining the overall intensity. Since we can't see what the actual color for (figure above) is, let's see what color each of these methods yields (figure below).

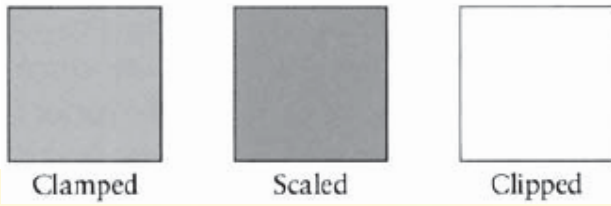


Figure 4: The results of three strategies for dealing with the same oversaturated color.

As we can see, we get three very different results. In terms of perceived color, the scaled is probably the closest though it's darker than the actual color values. If we weren't interested in the color but more in terms of saturation, then the clipped color is closer. Finally, the clamped value is what we get by default, and as you can see, the green component is biased down so that we lose a good sense of the "greenness" of the color we were trying to create.



Lecture No.24 Lighting II

Clamping Color Values

Now it's perfectly fine to end up with an oversaturated color and pass this result along to the graphics engine. What happens in the pipeline is an implicit clamping of the color values. Any value that's greater than one is clamped to one, and any less than zero are clamped to zero. So this has the benefit of requiring no effort on the part of the shader (*technology that is being used today for lighting and shading supported by Graphics hardware*) writer. Though this may make the rendering engine happy, it probably isn't what we want. Intuitively, we'd think that shining orange and green lights on a white surface would yield a strong green result. But letting the hardware clamp eradicates any predominant effect from the green light. Clamping is fast, but it tends to lose fidelity in the scene, particularly in areas where we would want and expect subtle changes as the light intensities interact, but end up with those interactions getting eradicated because the differences are all getting clamped out by the graphics hardware.

Scaling Color Values by Intensity

Instead of clamping, we might want to scale the color by dividing by the largest color value, thus scaling the **rgb** values into the $[0,1]$ range. In the example from figure 1, the final color values were $[1.0, 1.49, 0.49]$ meaning our largest color value was the green, at 1.49. Using this approach, we divide each element by 1.49, yielding a scaled color of $[0.671, 1.0, 0.329]$. Thus any values greater than one are scaled to one, while any other values are also scaled by the same amount. This maintains the hue and saturation but loses the intensity. This might not be acceptable because the contrast with other colors is lost, since contrast perception is nonlinear and we're applying a linear scaling. By looking at the three results, we can see the large difference between the resulting colors.

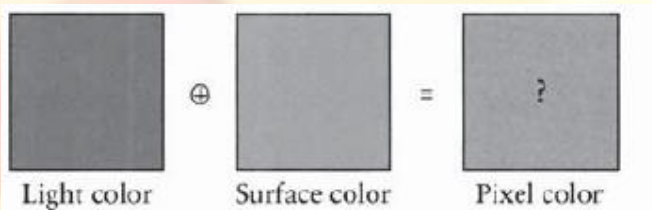


Figure 1: Adding colors can result in colors that are outside the displayable range.

Shifting Color Values to Maintain Saturation

One problem with clamping or scaling colors is that they get darker (lose saturation). An alternative to scaling is to maintain saturation by shifting color values. This technique is called clipping, and it's a bit more complicated than color scaling or clamping. The idea is to create a gray-scale vector that runs along the black-white axis of the color cube that's got the same brightness as the original color and then to draw a ray at right angles to this vector that intersects (i.e., clips) the original color's vector. We need to check to make sure that the grayscale vector is itself within the $[0,1]$ range and then to check the sign of the ray elements to see if the color elements need to be increased or decreased. As we are probably wondering, this can result in adding in a color value that wasn't in the original color, but this is a direct result of wanting to make sure that the overall brightness is the

same as the original color. And, of course, everything goes to hell in a handbasket if we've got overly bright colors, which leave us with decisions about how to nudge the gray-scale vector into the $[0,1]$ range, since that means you can't achieve the input color's saturation value. Then we're back to clamping or scaling again.

ColorSpace Tool

The ColorSpace tool is a handy tool that we can use to interactively add two colors together to see the effects of the various strategies for handling oversaturated colors. We simply use the sliders to select the **rgb** color values for each color. The four displays in Figure 5 show the composite, unmodified values of the resulting color (with no color square) and the clamped, clipped, and scaled color **rgb** values along with a color square illustrating those color values.

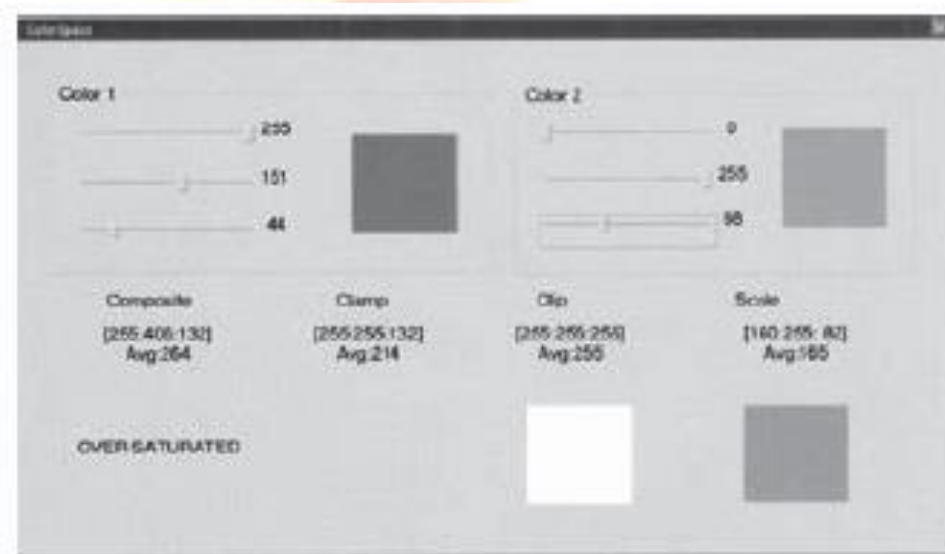


Figure 2: The ColorSpace tool interface.

Negative Colors and Darklights

We may be wondering, if we can have color values greater than the range in intermediate calculations, can we have negative values? Yes, we can! They are called "darklights" after their description in an article [GLASSNER 1992] in *Graphic Gems III*. Since this is all just math until we pass it back to the graphics hardware, we can pretty much do anything we want, which is pretty much the idea behind programmable shaders (*technology used by today graphics hardware for lighting and shading*)! Darklights are nothing more than lights in which one or more of the color values are negative. Thus instead of contributing to the overall lighting in a scene, we can specify a light that diminishes the overall lighting in a scene. Darklights are used to eliminate bright areas when we're happy with all the lighting in your scene *except* for an overly bright area. Darklights can also be used to affect a scene if we want to filter out a specific **rgb** color. If we wanted to get a *night vision* effect, we could use a darklight with negative red and blue values, for example, which would just leave the green channel.

Alpha Blending

Up to this point, we've been fairly dismissive of the mysterious alpha component that rides along in all of the color4 structure. Now, we may finally learn its dark secrets. A lot of power is hidden away inside the alpha component.

Loosely, the alpha component of the RGBA quad represents the opaqueness of a surface. An alpha value of 0xFF (255) means the color is completely opaque, and an alpha value of 0x00 (0) means the color is completely transparent. Of course, the value of the alpha component is fairly meaningless unless we actually activate the alpha blending step. If we want, we can set things up a different way, such as having 0x00(0) mean that the color is completely opaque. The meaning of alpha is dependent on how we set up the alpha blending step.

As you rasterize primitives, each pixel that we wish to change in the frame buffer gets sent through the alpha blending step. That pixel is combined using blending factors to the pixel that is currently in the frame buffer. We can add the two pixels together, multiply them together, linearly combine them using the alpha component, and so forth. The name "alpha blending" comes from the fact that generally the blending factors used are either the alpha or the inverse of the alpha.

The Alpha Blending Equation

The equation that governs the behavior of the blending is defined as follows:

$$\text{final color} = \text{source} \times \text{source blend factor} + \text{destination} \times \text{destination blend factor}$$

Final color is the color that goes to the frame buffer after the blending operation. Source is the pixel we are attempting to draw to the frame buffer, generally one of the many pixels in a triangle we have to draw. Destination is the pixel that already exists in the frame buffer before we attempt to draw a new one. The source and destination blend factors are variables that modify how the colors are combined together. The blend factors are the components we have control over in the equation; we cannot modify the positions of any of the terms or modify the operations performed on them.

For example, say we want an alpha blending equation to do nothing—to just draw the pixel from the triangle and not consider what was already there at all. An equation that would accomplish this would be:

$$\text{final color} = \text{source} \times 1.0 + \text{destination} \times 0.0$$

As we can see, the destination-blending factor is 0 and the source-blending factor is 1. This reduces the equation to:

$$\text{final color} = \text{source}$$

A second example would be if we wanted to multiply the source and destination components together before writing them to the frame buffer. This initially would seem difficult, as in the above equation they are only added together. However, the blending factors defined need not be constants; they can in fact be actual color components (or inverses thereof). The equation setup would be:

$$\text{final color} = \text{source} \times 0.0 + \text{destination} \times \text{source}$$

In this equation, the destination blend factor is set to the source color itself. Also, since the source blend factor is set to zero, the left-hand side of the equation drops away and we are left with:

final color = destination × source

Code Example

//This code will blend one image into the second

```

struct COLOR3{
    BYTE b;
    BYTE g;
    BYTE r;
};

//Assume the two bitmaps have the same size

COLOR3 *p1; //pointer to first bitmap;
COLOR3 *p2; //pointer to Second bitmap;

int k=0;

now we compute the blending factor av.

float av=(float)(alpha&255)/255;// alphaValue in floating point

compute the new pixel by using the alpha blending formula.

for(int i=firstBMP.Height; i>0; i--)
{
    for(int j=0; j<firstBMP.Width; j++)
    {
        get the red color
        p1->r =(BYTE)(( p1->r * (av))+ p2->r*(1.0f-(av)) );
        get the green color
        p1->g =(BYTE)(( p1->g * (av))+ p2->g*(1.0f-(av)) );
        get the blue color
        p1->b =(BYTE)(( p1->b * (av))+ p2->b*(1.0f-(av)) );

        p1++;
        p2++;
    }
}

```

send the new bitmap to display device

```
BlitData(displaydeviceContext, 0,0,firstImage.Width,firstImage.Height);
```

Following images shows the result of the above code.



Figure 3: First Image



Figure 4: Second Image



Figure 5: Blended image

Lecture No.25 Mathematics of Lighting and Shading Part I

LIGHTS AND MATERIALS

In order to understand how an object's color is determined, we'll need to understand the parts that come into play to create the final color. First, we need a source of illumination, typically in the form of a light source in our scene. A light has the properties of color (an **rgb** value) and intensity. Typically, these are multiplied to give scaled **rgb** values. Lights can also have attenuation, which means that their intensity is a function of the distance from the light to the surface. Lights can additionally be given other properties such as a shape (e.g., spotlights) and position (local or directional), but that's more in the implementation rather than the math of lighting effects. Given a source of illumination, we'll need a surface on which the light will shine. Here's where we get interesting effects. Two types of phenomena are important lighting calculations.

The first is the interaction of light with the surface boundary, and the second is the effect of light as it gets absorbed, transmitted, and scattered by interacting with the actual material itself. Since we really only have tools for describing surfaces of objects and not the internal material properties, light—surface boundary interactions are the most common type of calculation we'll see used, though we can do some interesting simulations of the interaction of light with material internals.

Materials are typically richer in their descriptions in an effort to mimic the effects seen in real light—material surface interactions. Materials are typically described using two to four separate colors in an effort to catch the nuances of real-world light—material surface interactions. These colors are the ambient, diffuse, specular, and emissive colors, with ambient and specular frequently grouped together, and emissive specified only for objects that generate light themselves. The reason there are different colors is to give different effects arising from different environmental causes. The most common lights are as follows:

Ambient lighting:

It is the overall color of the object due to the global ambient light level. This is the color of the object when there's no particular light, just the general environmental illumination. That is, the ambient light is an approximation for the global illumination in the environment, and relies upon no light in the scene. It's usually a global value that's added to every object in a scene.

Diffuse lighting:

It is the color of the object due to the effect of a particular light. The diffuse light is the light of the surface if the surface were perfectly matte. The diffuse light is reflected in all directions from the surface and depends only on the angle of the light to the surface normal.

Specular lighting:

It is the color of the highlights on the surface. The specular light mimics the shininess of a surface, and its intensity is a function of the light's reflection angle off the surface.

Emissive lighting:

When we need an object to "glow" in a scene, we can do this with an emissive light. This is just an additional color source added to the final light of the object. Because we're simulating an object giving off its own light; we'd still have to add a real "light" to get an effect on objects in a scene.

Before we get into exactly what these types of lighting are, let's put it in perspective for our purpose of writing shader code. Shading is simply calculating the color reflected off a surface (which is pretty much what shaders do). When a light reflects off a surface, the light colors are modulated by the surface color (typically, the diffuse or ambient surface color). Modulation means multiplication, and for colors, since we are using **rgb** values, this means component-by-component multiplication. So for light source l with color (r_l, g_l, b_l) shining on surface s with color (r_s, g_s, b_s) , the resulting color r would be:

$$r = l \otimes s$$

or, multiplying it out, we get

$$r = [l_r s_r, l_g s_g, l_b s_b]$$

Where the resulting rgb values of the light and surface are multiplied out to get the final color's rgb values

The final step after calculating all the lighting contributions is to add together all the lights to get the final color. So a shader might typically do the following:

1. Calculate the overall ambient light on a surface.
2. For each light in a scene, calculate the diffuse and specular contribution for each light.
3. Calculate any emissive light for a surface.
4. Add all these lights together to calculate the final color value.

In the real world, we get some sort of interaction (reflection, etc.) when a photon interacts with a surface boundary. Thus we see the effects not only when we have a transparent—opaque boundary (like airplastic), but also a transparent—transparent boundary (like air-water). The key feature here is that we get some visual effect when a photon interacts with some boundary between two different materials. The conductivity of the materials directly affects how the photon is reflected. At the surface of a conductor (metals, etc.), the light is mostly reflected. For dielectrics (nonconductors), there is usually more penetration and transmittance of the light. For both kinds of materials, the dispersion of the light is a function of the roughness of the surface (Figure 1 and 2).

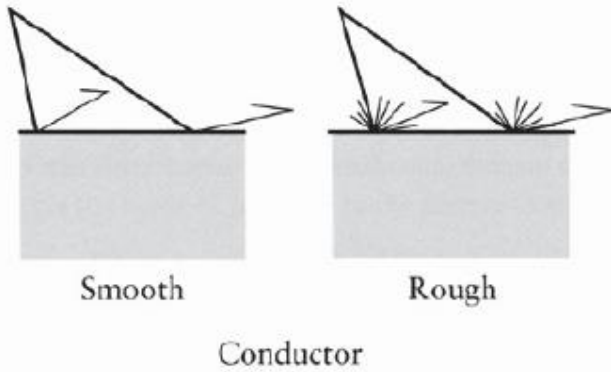


Figure 1: Light reflecting from a rough and smooth surface of a conductor.

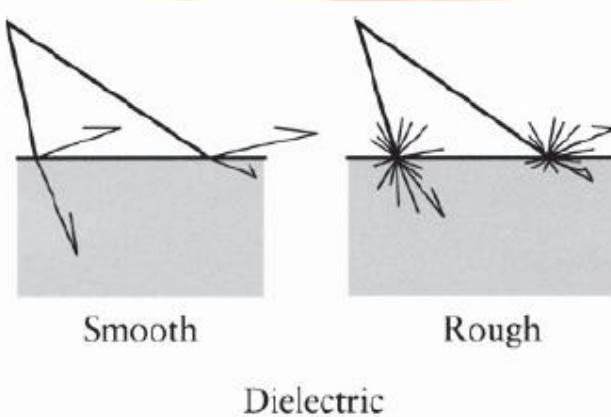


Figure 2: Light reflecting from a rough and smooth surface of a dielectric showing some penetration.

The simplest model assumes that the roughness of the surface is so fine that light is dispersed equally in all directions as shown in Figure 1, though later we'll look at fixing this assumption. A generalization is that conductors are opaque and dielectrics are transparent. This gets confusing since most of the dielectric surfaces that we are interested in modeling are mixtures and don't fall into the simple models we've described so far. Consider a thick colored lacquer surface. The lacquer itself is transparent, but suspended in the lacquer are reflective pigment off of which light gets reflected, bounced, split, shifted or altered before perhaps reemerging from the surface. This can be seen in Figure 3, where the light rays are not just reflected but bounced around a bit inside the medium before getting retransmitted to the outside.

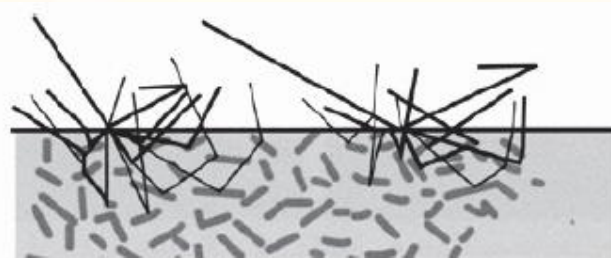


Figure 3: Subsurface scattering typical of pigment-saturated translucent coatings.

Metallic paint, brushed metal, velvet, etc. are all materials for which we need to examine better models to try to represent these surfaces. But with a little creativity in the modeling, it's possible to mimic the effect. Figure 4 shows what you get when you use multiple broad specular terms for multiple base colors combined with a more traditional shiny specular term. There's also a high-frequency normal perturbation that simulates the sparkle from a metallic flake pigment. As we can see, we can get something that looks particularly striking with a fairly simple model.

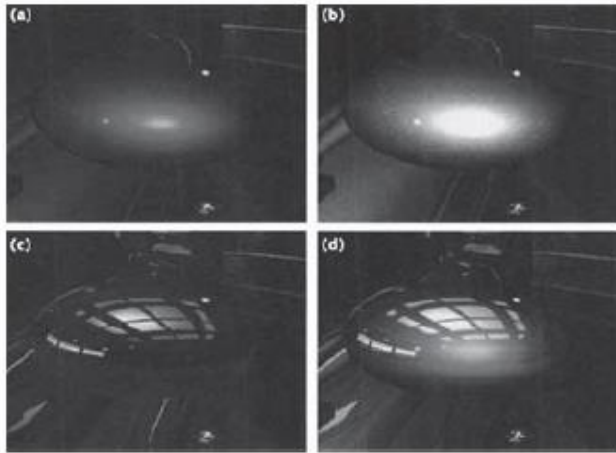


Figure 4: A simple shader to simulate metallic paint: (a) shows the two-tone paint shading pass; (b) shows the specular sparkle shading pass; (c) shows the environment mapping pass; (d) shows the final composite image

The traditional model gives us a specular term and a diffuse term. We have been able to add in texture maps to give our scenes some uniqueness, but the lighting effects have been very simple. Shaders allow us to be much more creative with lighting effects. As Figure 4 shows, with just a few additional specular terms, we can bring forth a very interesting look. But before we go off writing shaders, we'll need to take a look at how it all fits together in the graphics pipeline. And a good place to start is by examining the traditional lighting model.

Lecture No.26 Mathematics of Lighting and Shading Part II Light Types and Shading Models

Light Types

Now that we have a way to find the light hitting a surface, we're going to need some lights! There are three types of lights we are going to discuss.

I. Parallel Lights (or Directional Lights)

Parallel lights cheat a little bit. They represent light that comes from an infinitely far away light source. Because of this, all of the light rays that reach the object are parallel (hence the name). The standard use of parallel lights is to simulate the sun. While it's not infinitely far away, 93 million miles is good enough!

The great thing about parallel lights is that a lot of the math goes away. The attenuation factor is always 1 (for point/spotlights, it generally involves divisions if not square roots). The incoming light vector for calculation of the diffuse reflection factor is the same for all considered points, whereas point lights and spotlights involve vector subtractions and a normalization per vertex.

Typically, lighting is the kind of effect that is sacrificed for processing speed. Parallel light sources are the easiest and therefore fastest to process. If we can't afford to do the nicer point lights or spotlights, falling back to parallel lights can keep our frame rates at reasonable levels.

II. Point Lights

Point lights are one step better than directional lights. They represent infinitesimally small points that emit light. Light scatters out equally in all directions. Depending on how much effort we're willing to expend on the light, we can have the intensity falloff based on the inverse squared distance from the light, which is how real lights work.

$$\text{attenuation_factor} = \frac{k}{|\text{surface_location} - \text{light_location}|^2}$$

The light direction is different for each surface location (otherwise the point light would look just like a directional light). The equation for it is:

$$\text{light_direction} = \frac{\text{surface_location} - \text{light_location}}{|\text{surface_location} - \text{light_location}|}$$

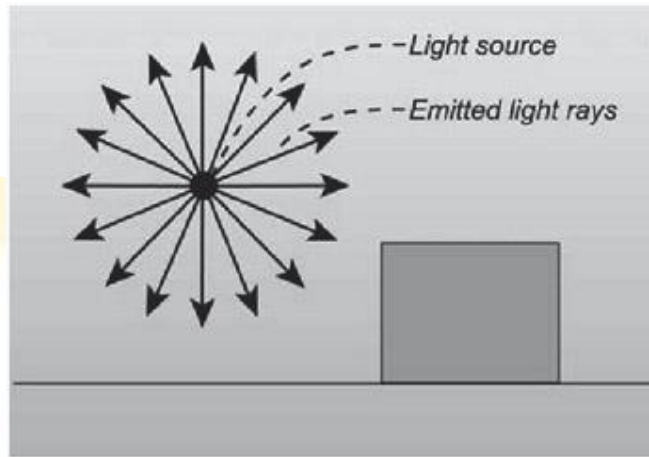


Figure 1: Point light sources

III. Spotlights

Spotlights are the most expensive type of light we discuss in this course and should be avoided if possible because it is not for real time environment. We model a spotlight not unlike the type we would see in a theatrical production. They are point lights, but light only leaves the point in a particular direction, spreading out based on the aperture of the light.

Spotlights have two angles associated with them. One is the internal cone whose angle is generally referred to as theta (θ). Points within the internal cone receive all of the light of the spotlight; the attenuation is the same as it would be if point lights were used. There is also an angle that defines the outer cone; the angle is referred to as phi. Points outside the outer cone receive no light. Points outside the inner cone but inside the outer cone receive light, usually a linear falloff based on how close it is to the inner cone.

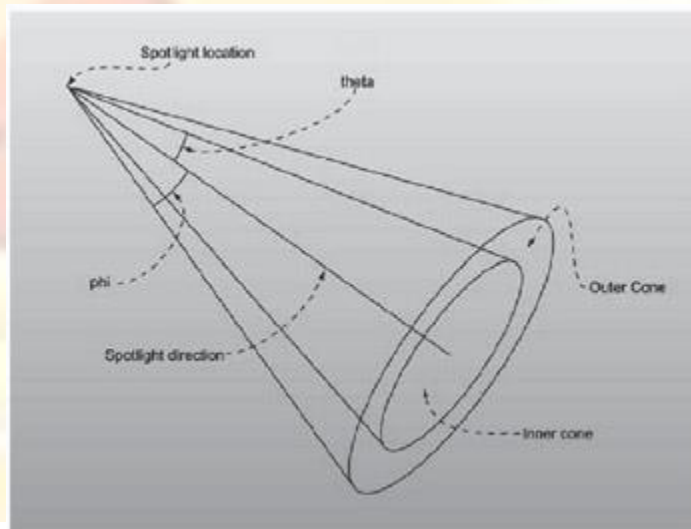


Figure 2: A spotlight

If we think all of this sounds mathematically expensive, we're right. Some library packages like OpenGL and Direct3D implements lighting for us, so we won't need to worry about the implementation of the math behind spotlights, but rest assured that they're extremely expensive and can slow down our graphics application a great deal.

Then again, they do provide an incredible amount of atmosphere when used correctly, so we will have to figure out a line between performance and aesthetics.

Shading Models

Once we've found basic lighting information, we need to know how to draw the triangles with the supplied information. There are currently three ways to do this; the third has just become a hardware feature with DirectX 9.0 In our previous lectures we have already studied flat and gouraud shading triangle algorithms.

I. Lambert

Triangles that use Lambertian shading are painted with one solid color instead of using a gradient. Typically each triangle is lit using that triangle's normal. The resulting object looks very angular and sharp. Lambertian shading was used mostly back when computers weren't fast enough to do Gouraud shading in real time. To light a triangle, you compute the lighting equations using the triangle's normal and any of the three vertices of the triangle.

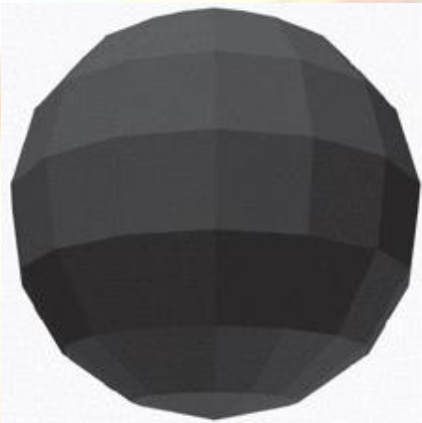


Figure 3: Flat shaded view of our polygon mesh

II. Gouraud

Gouraud (pronounced *garrow*) shading is the current de facto shading standard in accelerated 3D hardware. Instead of specifying one color to use for the entire triangle, each vertex has its own separate color. The color values are linearly interpolated across the triangle, creating a smooth transition between the vertex color values. To calculate the lighting for a vertex, we use the position of the vertex and a vertex normal.

Of course, it's a little hard to correctly define a normal for a vertex. What people do instead is average the normals of all the polygons that share a certain vertex, using that as the vertex normal. When the object is drawn, the lighting color is found for each vertex (rather than each polygon), and then the colors are linearly interpolated across the object. This creates a slick and smooth look, like the one in Figure 4.



Figure 4: Gouraud shaded view of our polygon mesh

One problem with Gouraud shading is that the triangles' intensities can never be greater than the intensities at the edges. So if there is a spotlight shining directly into the center of a large triangle, Gouraud shading will interpolate the intensities at the three dark corners, resulting in an incorrectly dark triangle. The internal highlighting problem usually isn't that bad. If there are enough triangles in the model, the interpolation done by Gouraud shading is usually good enough. If we really want internal highlights but only have Gouraud shading, we can subdivide the triangle into smaller pieces.

III. Phong

Phong shading is the most realistic shading model we are going to talk about, and also the most computationally expensive. It tries to solve several problems that arise when we use Gouraud shading. If we're looking for something more realistic, some authors have also discussed nicer shading models like Torrance-Sparrow, but they aren't real time (at least not right now). First of all, Gouraud shading uses a linear gradient. Many objects in real life have sharp highlights, such as the shiny spot on an apple. This is difficult to handle with pure Gouraud shading. The way Phong does this is by interpolating the normal across the triangle face, not the color value, and the lighting equation is solved individually for each pixel.

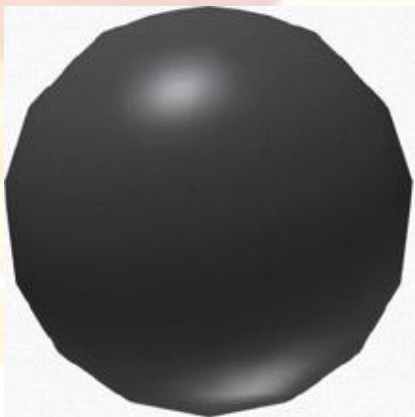


Figure 5: Phong shaded view of a polygon mesh

Phong shading isn't technically supported in hardware. But we can now program our own Phong rendering engine, and many other special effects, using shaders, a hot new technology.

Lecture No.27 Review II

27.1 CLIPPING - Concept

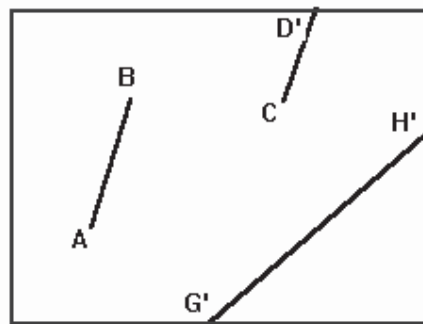
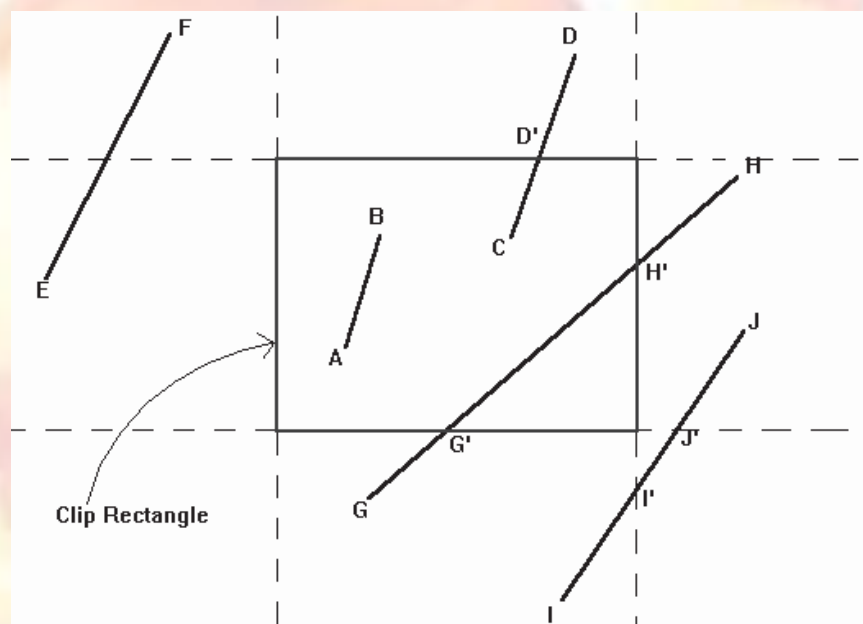
It is desirable to restrict the effect of graphics primitives to a sub-region of the canvas, to protect other portions of the canvas. All primitives are clipped to the boundaries of this **clipping rectangle**; that is, primitives lying outside the clip rectangle are not drawn.

The default clipping rectangle is the full canvas (the screen), and it is obvious that we cannot see any graphics primitives outside the screen.

A simple example of line clipping can illustrate this idea:

This is a simple example of line clipping: the display window is the canvas and also the default clipping rectangle, thus all line segments inside the canvas are drawn.

The red box is the clipping rectangle we will use later, and the dotted line is the extension of the four edges of the clipping rectangle.



27.2 Point Clipping

Assuming a rectangular clip window, point clipping is easy. we save the point if:

$$X_{\min} \leq x \leq X_{\max}$$

$$y_{\min} \leq y \leq y_{\max}$$

27.3 Line Clipping

This section treats clipping of lines against rectangles. Although there are specialized algorithms for rectangle and polygon clipping, it is important to note that other graphic primitives can be clipped by repeated application of the line clipper.

27.4 Cohen-Sutherland algorithm - Conclusion

In summary, the Cohen-Sutherland algorithm is efficient when out-code testing can be done cheaply (for example, by doing bit-wise operations in assembly language) and trivial acceptance or rejection is applicable to the majority of line segments. (For example, large windows - everything is inside, or small windows - everything is outside).

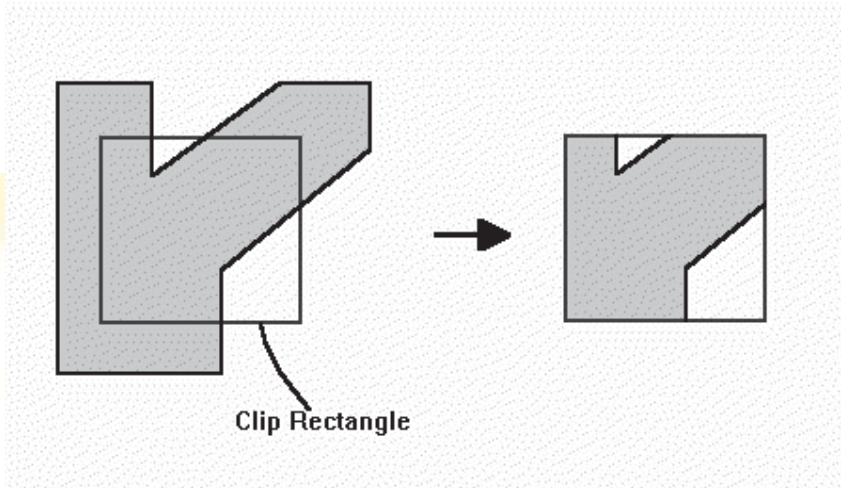
27.5 Liang-Barsky Algorithm - Conclusion

In general, the Liang_Barsky algorithm is more efficient than the Cohen_Sutherland algorithm, since intersection calculations are reduced. Each update of parameters u_1 and u_2 requires only one division; and window intersections of the line are computed only once, when the final values of u_1 and u_2 have computed. In contrast, the Cohen-Sutherland algorithm can repeatedly calculate intersections along a line path, even though the line may be completely outside the clip window, and, each intersection calculation requires both a division and a multiplication. Both the Cohen_Sutherland and the Liang_Barsky algorithms can be extended to three-dimensional clipping.

27.6 Polygon Clipping

A polygon is usually defined by a sequence of vertices and edges. If the polygons are unfilled, line-clipping techniques are sufficient however, if the polygons are filled, the process is more complicated. A polygon may be fragmented into several polygons in the clipping process, and the original colour associated with each one. The Sutherland-Hodgeman clipping algorithm clips any polygon against a convex clip polygon. The Weiler-Atherton clipping algorithm will clip any polygon against any clip polygon. The polygons may even have holes.

The following example illustrates a simple case of polygon clipping.



27.7 Sutherland and Hodgman's polygon-clipping algorithm:-

Sutherland and Hodgman's polygon-clipping algorithm uses a divide-and-conquer strategy: It solves a series of simple and identical problems that, when combined, solve the overall problem. The simple problem is to clip a polygon against a single infinite clip edge. Four clip edges, each defining one boundary of the clip rectangle, successively clip a polygon against a clip rectangle.

Note the difference between this strategy for a polygon and the Cohen-Sutherland algorithm for clipping a line: The polygon clipper clips against four edges in succession, whereas the line clipper tests the outcode to see which edge is crossed, and clips only when necessary.

27.8 Steps of Sutherland-Hodgman's polygon-clipping algorithm

- Polygons can be clipped against each edge of the window one at a time. Windows/edge intersections, if any, are easy to find since the X or Y coordinates are already known.
- Vertices which are kept after clipping against one window edge are saved for clipping against the remaining edges.
- Note that the number of vertices usually changes and will often increase.

We are using the Divide and Conquer approach.

27.9 Shortcoming of Sutherlands -Hodgeman Algorithm

Convex polygons are correctly clipped by the Sutherland-Hodgeman algorithm, but concave polygons may be displayed with extraneous lines. This occurs when the clipped polygon should have two or more separate sections. But since there is only one output vertex list, the last vertex in the list is always joined to the first vertex. There are several things we could do to correct display concave polygons. For one, we could split the concave polygon into two or more convex polygons and process each convex polygon separately.

Another approach to check the final vertex list for multiple vertex points along any clip window boundary and correctly join pairs of vertices. Finally, we could use a more general polygon clipper, such as with the Weiler-Atherton algorithm or the Weiler algorithm described in the next section.

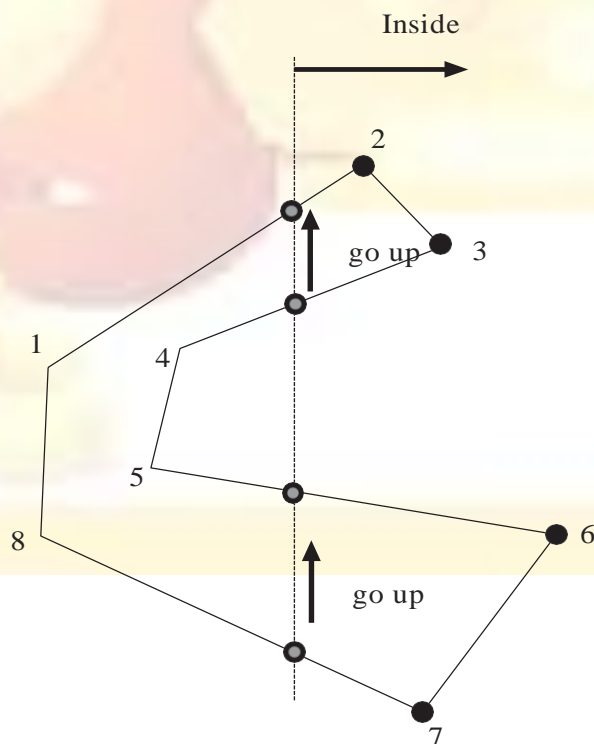
27.10 Weiler-Atherton Polygon Clipping

In this technique, the vertex-processing procedures for window boundaries are modified so that concave polygons are displayed correctly. This clipping procedure was developed as a method for identifying visible surfaces, and so it can be applied with arbitrary polygon-clipping regions.

The basic idea in this algorithm is that instead of always proceeding around the polygon edges as vertices are processed, we sometimes want to follow the window boundaries. Which path we follow depends on the polygon-processing direction (clockwise or counterclockwise) and whether the pair of polygon vertices currently being processed represents an outside-to-inside pair or an inside-to-outside pair. For clockwise processing of polygon vertices, we use the following rules:

- For an outside-to-inside pair of vertices, follow the polygon boundary
- For an inside-to-outside pair of vertices, follow the window boundary in a clockwise direction

In following figure, the processing direction in the Weiler-Atherton algorithm and the resulting clipped polygon is shown for a rectangular clipping window.



27.11 3D Concepts

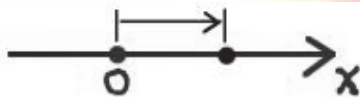
27.12 Coordinate Systems

Coordinate systems are the measured frames of reference within which geometry is defined, manipulated and viewed. In this system, you have a well-known point that serves as the origin (reference point), and three lines (axes) that pass through this point and are orthogonal to each other (at right angles – 90 degrees).

With the Cartesian coordinate system, you can define any point in space by saying how far along each of the three axes you need to travel in order to reach the point if you start at the origin.

Following are three types of the coordinate systems.

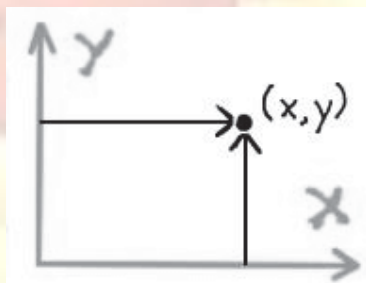
27.13 1-D Coordinate Systems:



This system has the following characteristics:

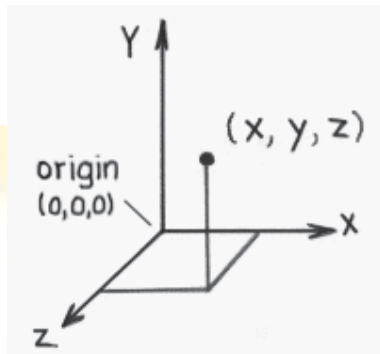
- Direction and magnitude along a single axis, with reference to an origin
- Locations are defined by a single coordinate
- Can define points, segments, lines, rays
- Can have multiple origins (frames of reference) and transform coordinates among them

27.14 2-D Coordinate Systems:



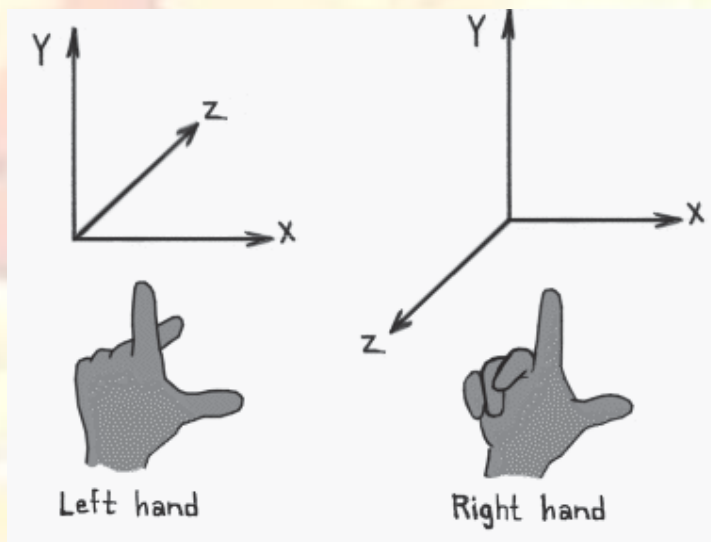
- Direction and magnitude along two axes, with reference to an origin
- Locations are defined by x, y coordinate pairs
- Can define points, segments, lines, rays, curves, polygons, (any planar geometry)
- Can have multiple origins (frames of reference and transform coordinates among them)

27.15 3-D Coordinate Systems:



- 3D Cartesian coordinate systems
- Direction and magnitude along three axes, with reference to an origin
- Locations are defined by x, y, z triples
- Can define cubes, cones, spheres, etc., (volumes in space) in addition to all one- and two-dimensional entities
- Can have multiple origins (frames of reference) and transform coordinates among them

27.16 Left-handed versus Right-handed



- Determines orientation of axes and direction of rotations
- Thumb = pos x, Index up = pos y, Middle out = pos z
- Most world and object axes tend to be right handed

- Left handed axes often are used for cameras

27.17 Right Handed Rule:

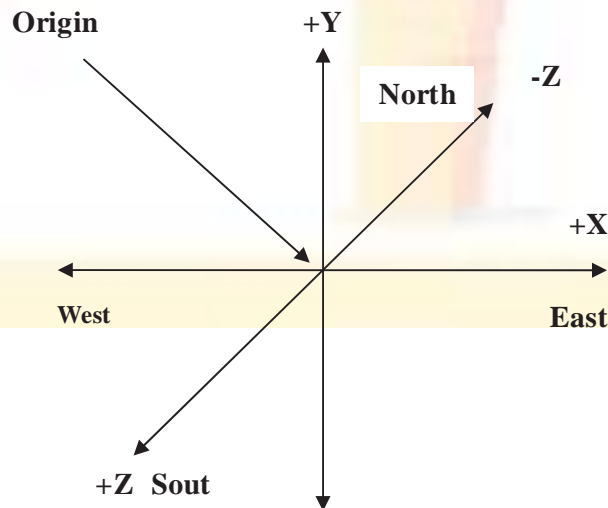
“Right Hand Rule” for rotations: grasp axis with right hand with thumb oriented in positive direction, fingers will then curl in direction of positive rotation for that



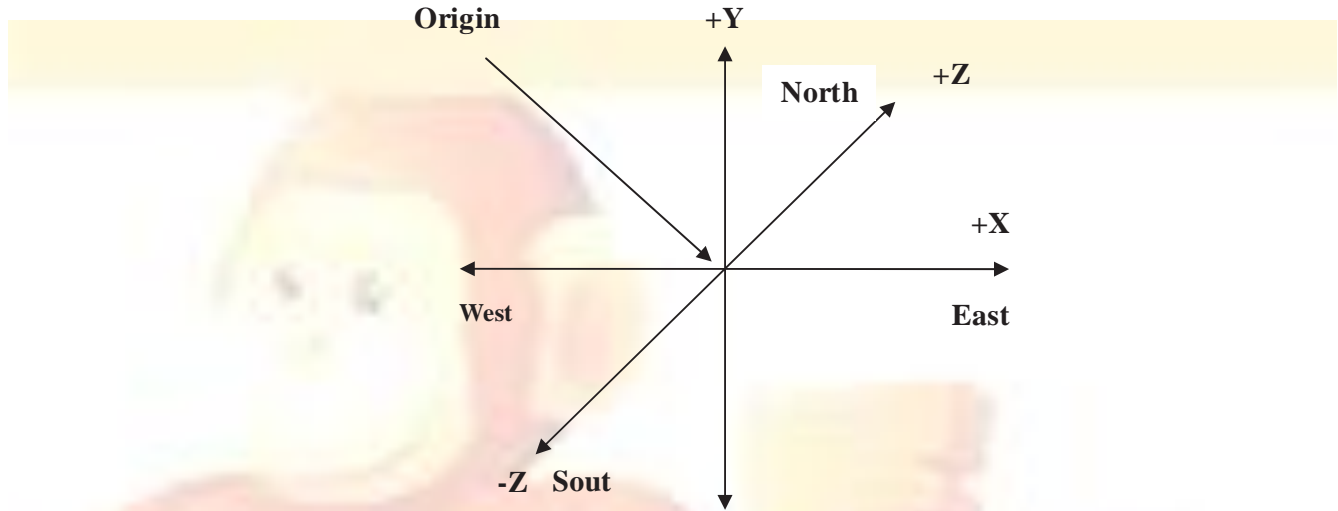
axis.

Right handed Cartesian coordinate system describes the relationship of the X,Y, and Z in the following manner:

- X is positive to the right of the origin, and negative to the left.
- Y is positive above the origin, and negative below it.
- Z is *negative* beyond the origin, and *positive* behind it.



27.18 Left Handed Rule:



Left handed Cartesian coordinate system describes the relationship of the X, Y and Z in the following manner:

- X is positive to the right of the origin, and negative to the left.
- Y is positive above the origin, and negative below it.
- Z is positive beyond the origin, and negative behind it.

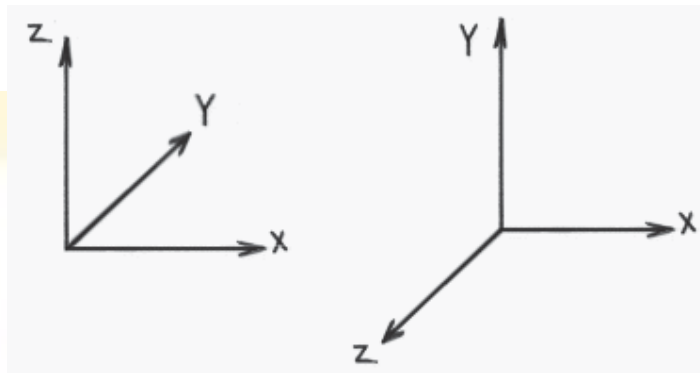
27.19 Defining 3D points in mathematical notations

3D points can be described using simple mathematical notations

$$P = (X, Y, Z)$$

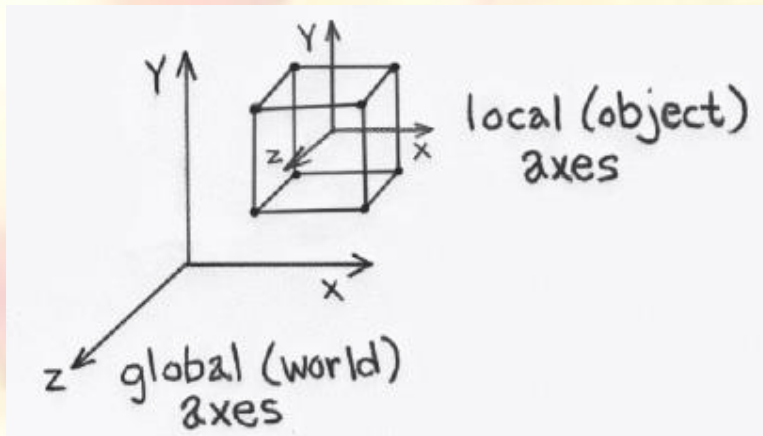
Thus the origin of the Coordinate system is located at point (0,0,0), while five units to the right of that position might be located at point (5,0,0).

27.20 Y-up versus Z-up:



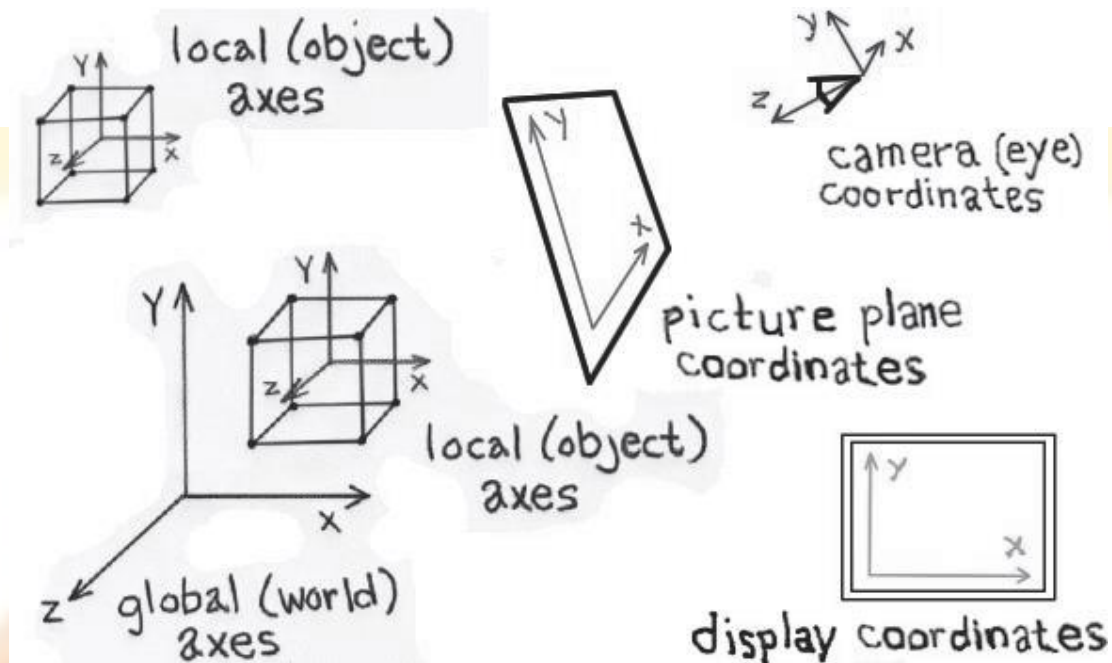
- z-up typically used by designers
- y-up typically used by animators
- orientation by profession supposedly derives from past work habits
- often handled differently when moving from application to application

27.21 Global and Local Coordinate Systems:



- Local coordinate systems can be defined with respect to global coordinate system
- Locations can be relative to any of these coordinate systems
- Locations can be translated or "transformed" from one coordinate system to another.

27.22 Multiple Frames of Reference in a 3-D Scene:



- In fact, there usually are multiple coordinate systems within any 3-D screen
- Application data will be transformed among the various coordinate systems, depending on what's to be accomplished during program execution
- Individual coordinate systems often are hierarchically linked within the scene

27.23 The Polar Coordinate System

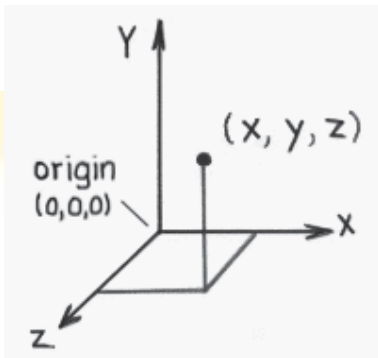
Cartesian systems are not the only ones we can use. We could have also described the object position in this way: “starting at the origin, looking east, rotate 38 degrees northward, 65 degrees upward, and travel 7.47 feet along this line. “As you can see, this is less intuitive in a real world setting. And if you try to work out the math, it is harder to manipulate (when we get to the sections that move points around). Because such polar coordinates are difficult to control, they are generally not used in 3D graphics.

27.24 Defining Geometry in 3-D

Here are some definitions of the technical names that will be used in 3D lectures.

Modeling: is the process of describing an object or scene so that we can construct an image of it.

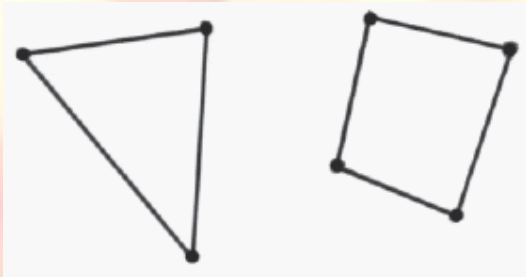
27.25 Points & polygons:



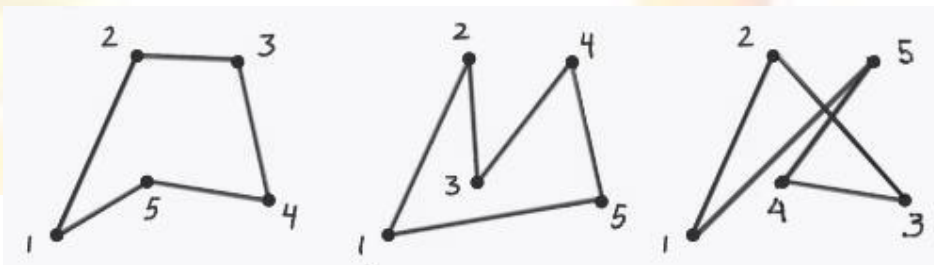
- Points: three-dimensional locations (or coordinate triples)



- Vectors: - have direction and magnitude; can also be thought of as displacement



- Polygons: - sequences of “correctly” co-planar points; or an initial point and a sequence of vectors



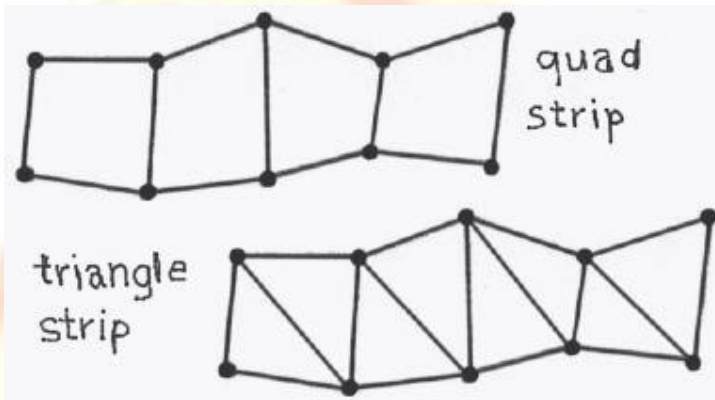
27.26 Primitives

Primitives are the fundamental geometric entities within a given data structure.

- We have already touched on point, vector and polygon primitives



- Regular Polygon Primitives - square, triangle, circle, n-polygon, etc.

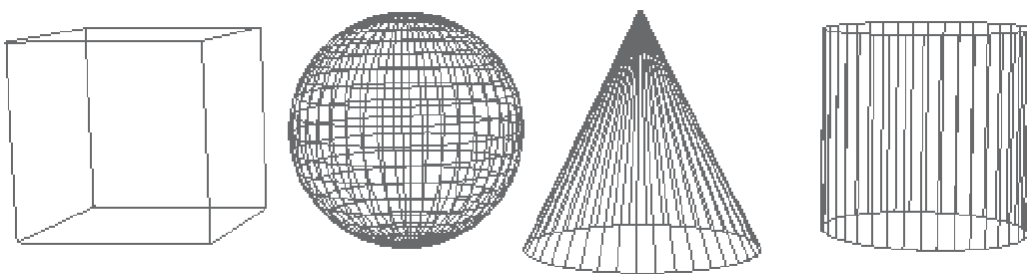


- Polygon strips or meshes
- Meshes provide a more economical description than multiple individual polygons

For example, 100 individual triangles, each requiring 3 vertices, would require 100×3 or 300 vertex definitions to be stored in the 3-D database.

By contrast, triangle strips require $n + 2$ vertex definitions for any n number of triangles in the strip. Hence, a 100 triangle strip requires only 102 unique vertex definitions.

- Meshes also provide continuity across surfaces which is important for shading calculations



- 3D primitives in a polygonal database

3D shapes are represented by polygonal meshes that define or approximate geometric surfaces.



- With curved surfaces, the accuracy of the approximation is directly proportional to the number of polygons used in the representation.
- More polygons (when well used) yield a better approximation.
- But more polygons also exact greater computational overhead, thereby degrading interactive performance, increasing render times, etc.

27.27 *Rendering*

- The process of computing a two dimensional image using a combination of a three-dimensional database, scene characteristics, and viewing transformations. Various algorithms can be employed for rendering, depending on the needs of the application.

27.28 *Tessellation*

- The subdivision of an entity or surface into one or more non-overlapping primitives. Typically, renderers decompose surfaces into triangles as part of the rendering process.

27.29 *Sampling*

- The process of selecting a representative but finite number of values along a continuous function sufficient to render a reasonable approximation of the function for the task at hand.

27.30 *Level of Detail (LOD)*

- To improve rendering efficiency when dynamically viewing a scene, more or less detailed versions of a model may be swapped in and out of the scene database depending on the importance (usually determined by image size) of the object in the current view.

27.31 *Transformations*

The process of moving points in space is called transformation.

27.32 Types of Transformation

There are various types of transformations as we have seen in case of 2D transformations. These include:

- Translation
- Rotation
- Scaling
- Reflection
- Shearing

a) Translation

Translation is used to move a point, or a set of points, linearly in space. Since now we are talking about 3D, therefore each point has 3 coordinates i.e. x, y and z. similarly, the translation distances can also be specified in any of the 3 dimensions. These Translation Distances are given by tx, ty and tz.

For any point P(x,y,z) after translation we have P'(x',y',z') where

$$x' = x + tx ,$$

$$y' = y + ty ,$$

$$z' = z + tz$$

and (tx, ty, tz) is Translation vector

Now this can be expressed as a single matrix equation:

$$P' = P + T$$

Where:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

$$P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

3D Translation Example

We may want to move a point “3 meters east, -2 meters up, and 4 meters north.” What would be done in such event?

Steps for Translation

Given a point in 3D and a translation vector, it can be translated as follows:

$$\text{Point3D point} = (0, 0, 0)$$

$$\text{Vector3D vector} = (10, -3, 2.5)$$

Adding vector to point

$$\text{point.x} = \text{point.x} + \text{vector.x};$$

$$\text{point.y} = \text{point.y} + \text{vector.y};$$

$$\text{point.z} = \text{point.z} + \text{vector.z};$$

And finally we have translated point.

Homogeneous Coordinates

Analogous to their 2D Counterpart, the homogeneous coordinates for 3D translation can be expressed as :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(t_x, t_y, t_z) \cdot P$$

On solving the RHS of the matrix equation, we get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Which shows that each of the 3 coordinates gets translated by the corresponding translation distance.

Rotation

Rotation is the process of moving a point in space in a non-linear manner

- ◆ We need to know three different angles:
- ◆ How far to rotate around the X axis (YZ rotation, or “pitch”)
- ◆ How far to rotate around the Y axis (XZ rotation, or “yaw”)
- ◆ How far to rotate around the Z axis (XY rotation, or “roll”)

Column vector representation:

$$P' = R \cdot P$$

Where

$$P' = \begin{bmatrix} x' \\ y' \end{bmatrix}$$

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix}$$

$$P = \begin{bmatrix} x \\ y \end{bmatrix}$$

Rotation: Homogeneous Coordinates

The rotation can now be expressed using homogeneous coordinates as:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = R(\theta) \cdot P$$

... Now in 3D

- ◆ Rotation can be about any of the three axes:
- ◆ About z-axis (i.e. in xy plane)
- ◆ About x-axis (i.e. in yz plane)
- ◆ About y-axis (i.e. in xz plane)

Roll : around z-axis

Pitch: around x-axis

Yaw: around y-axis

◆ Rotation about z-axis

(i.e. in xy plane):

$$\begin{aligned}x' &= x \cos\theta - y \sin\theta \\y' &= x \sin\theta + y \cos\theta \\z' &= z\end{aligned}$$

by Cyclic permutation

Rotation about x-axis

(i.e. in yz plane):

$$\begin{aligned}x' &= x \\y' &= y \cos\theta - z \sin\theta \\z' &= y \sin\theta + z \cos\theta\end{aligned}$$

and

Rotation about y-axis

(i.e. in xz plane):

$$\begin{aligned}x' &= z \sin\theta + x \cos\theta \\y' &= y \\z' &= z \cos\theta - x \sin\theta\end{aligned}$$

b) SCALING:-

◆ Coordinate transformations for scaling relative to the origin are

- ◆ $X' = X \cdot S_x$
- ◆ $Y' = Y \cdot S_y$
- ◆ $Z' = Z \cdot S_z$

Uniform Scaling

- ◆ We preserve the original shape of an object with a uniform scaling
- ◆ $(S_x = S_y = S_z)$

Differential Scaling

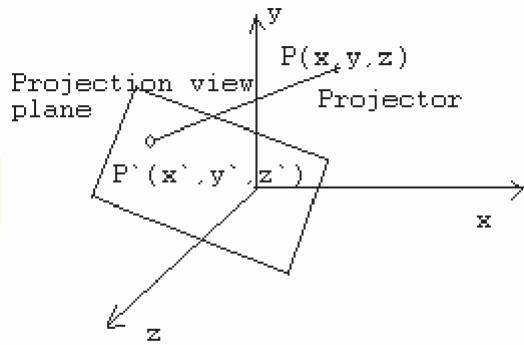
- ◆ We do not preserve the original shape of an object with a differential scaling
- ◆ $(S_x \neq S_y \neq S_z)$

Scaling w.r.t. Origin

$$\begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

27.33 PROJECTION

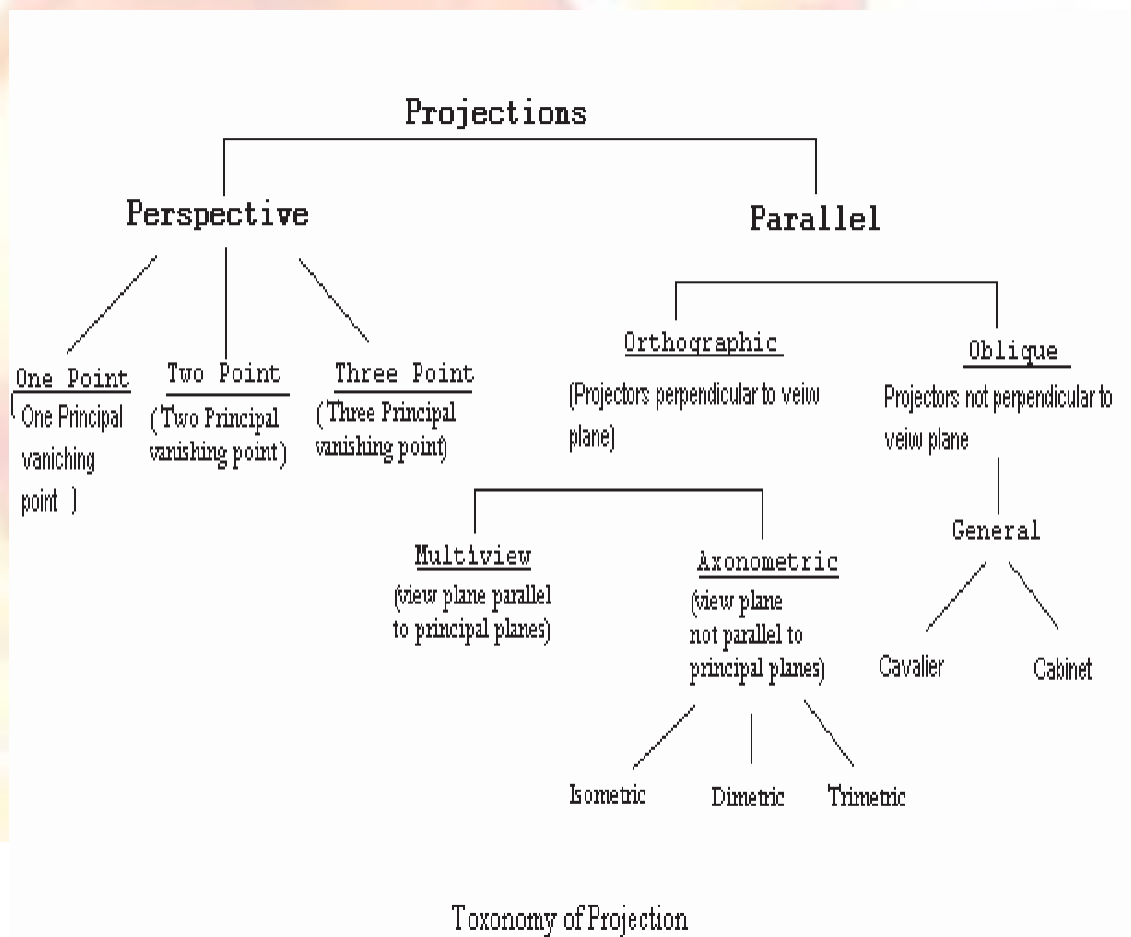
Projection can be defined as a mapping of point $P(x,y,z)$ onto its image $P'(x',y',z')$ in the projection plane or view plane, which constitutes the display surface



The Problem of Projection

Methods of Projection

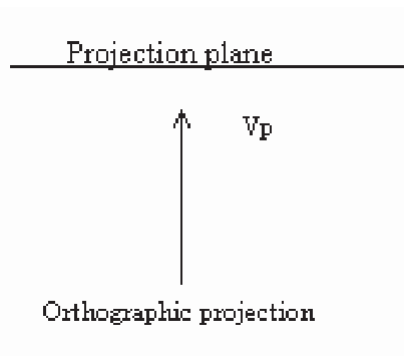
- Parallel Projection
- Perspective Projection



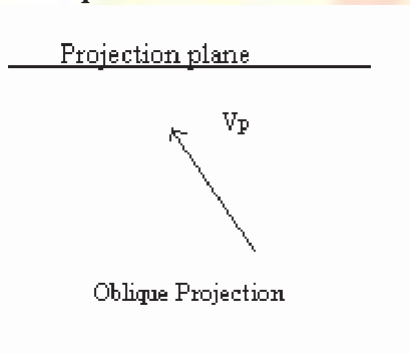
Taxonomy of Projection

Parallel Projection is divided into Orthographic and Oblique transformations.

- Orthographic



- **Oblique**

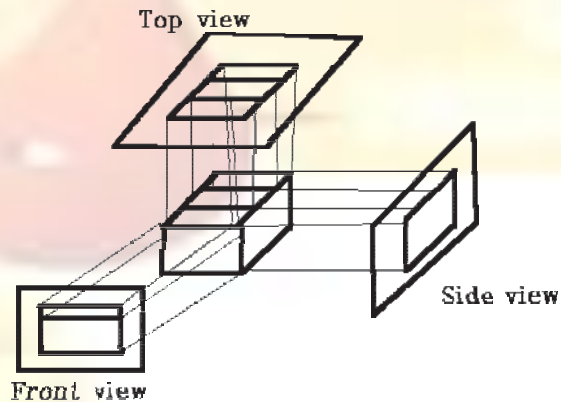


Orthographic projection has two types

- Multiview
- Axonometric

Multiview:

There are three orthographic views of an object.



Axonometric projections:

There are three axonometric projections:

- Isometric
- Dimetric
- Trimetric

1. Isometric

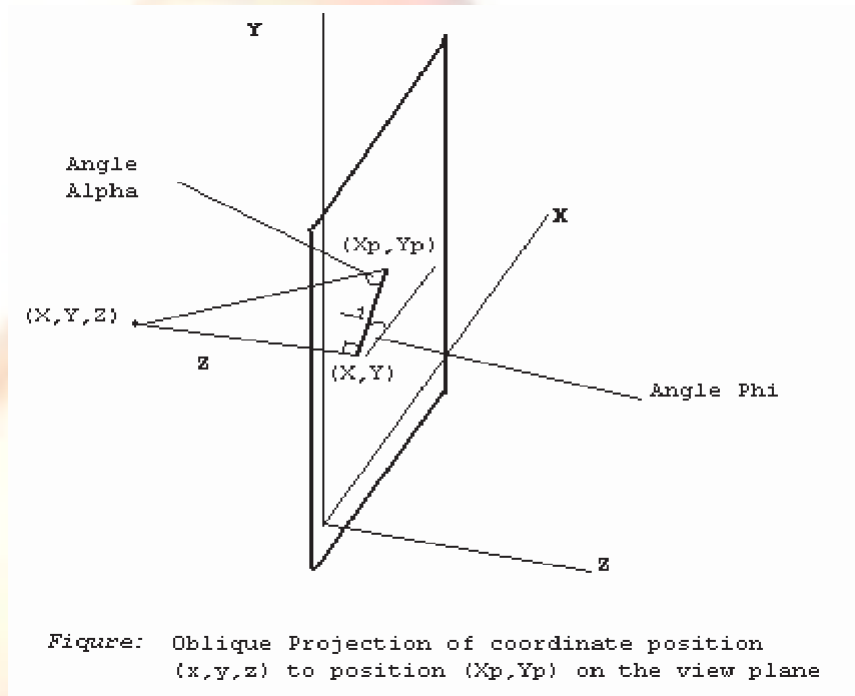
The projection plane intersects each coordinate axis in the model coordinate system at an equal distance or the direction of projection makes equal angles with all of the three principal axes

2. Dimetric

The direction of projection makes equal angles with exactly two of the principal axes

3. Trimetric

The direction of projection makes unequal angles with the three principal axes



$$\blacklozenge X_p = x + z (L_1 \cos(\Phi))$$

$$\blacklozenge Y_p = y + z (L_1 \sin(\Phi))$$

Where $L_1 = L/z$

Lecture No.28 Review III

28.1 *Perspective Projection*

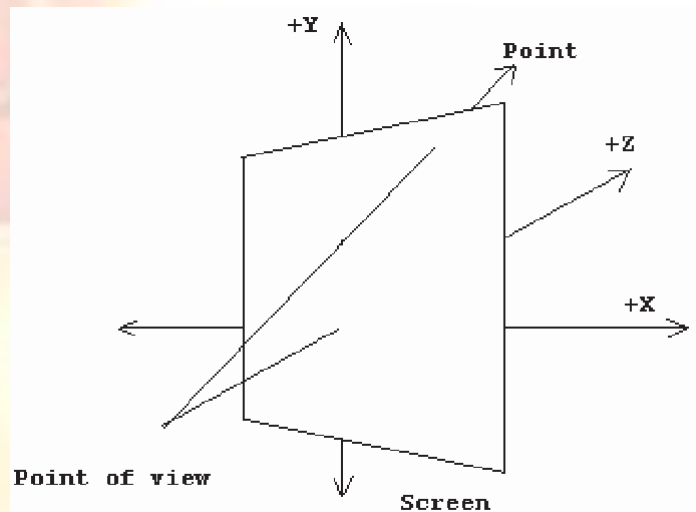
As opposed to parallel projection, perspective projection gives a more realistic view of the objects in the scene. The objects away from the POV are projected smaller whereas those nearby are projected to appear proportionately larger. The idea is that of looking at the scene through the projection plane / screen.

The POV represents the viewer's eye, and we presume that the viewer will be behind the center of the screen.

Note: We will use Left hand rule to describe 3D coordinate system.

Two common approaches are used with this;

1. The first approach is where the POV is at some point $(0, 0, -z)$ and the screen lies on the X-Y plane, graphically, this looks like figure given below:



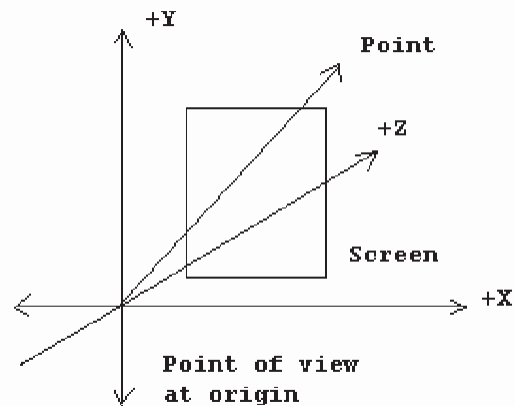
2. The second approach is where the POV lies at the origin, and the screen lies on a plane at some +z coordinate, as shown in figure given below:

As we will see later, this second approach is much more convenient when we add features making it possible for

the POV to move around the 3D world or for objects to move around in the world.

Calculating the screen pixel that correlates to a 3D point is now a matter of simple geometry. From a viewpoint above the screen and POV (looking at the X-Z plane), the geometry appears like the one shown in figure below:

In geometric terms, we say that the triangle from A to B to S is similar to the triangle from A to C to P because the three angles that make up the triangles are the same: the angle from AB to AS is the same as the angle from AC to AP, the two right angles are both 90 degrees, and therefore the remaining two angles are the same (the sum of the angles in a triangle is always 180 degrees). What also holds true from similar triangles is that the ratio of two sides holds between the similar triangles; this means that the ratio of BS to AB is the same as the ratio of CP to AC. But we know what AB is-it is Screen.z ! and we



know what AC is—it is point.z ! and we know what CP is—it is point.x ! therefore:

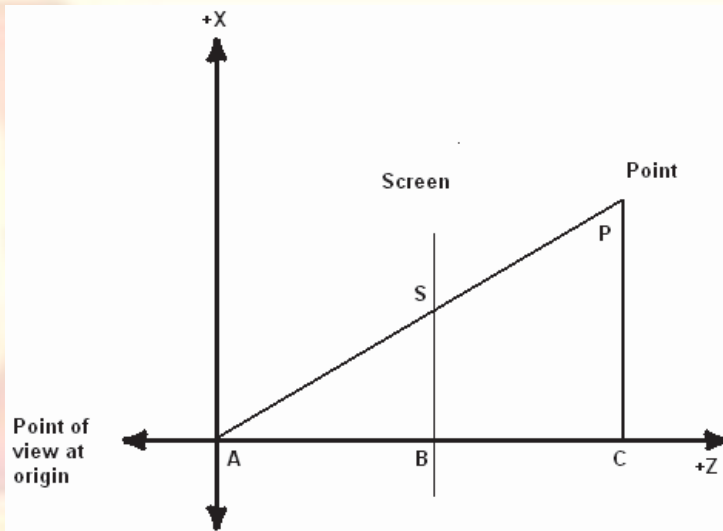
$$|BS| / |AB| = |CP| / |AC|$$

$$|BS| = |AB| * |CP| / |AC|$$

$$|BS| = \text{Screen.z} * \text{point.x} / \text{point.z}$$

Screen.z is the distance d from the point of view at origin or the scaling factor.

Notice that $|BS|$ is the length of the line segment that goes from B to S in world units. But we normally address the screen with the point (0,0) at the top left, with +X pixels moving to the right, and +Y pixels moving down—and not from the middle of the screen. And we draw to the screen in pixel units – not our world units (unless, of course, 1.0 in your world represents one pixel).



28.2 Triangles

Triangles are to 3D graphics what pixels are to 2D graphics. Every PC hardware accelerator under the sun uses triangles as the fundamental drawing primitive (well ... scan line aligned trapezoids actually, but that's a hardware implementation issue). When you draw a polygon, hardware devices really draw a fan of triangles. Triangles "flesh out" a 3D object, connecting them together to form a skin or mesh that defines the boundary surface of an object. Triangles, like polygons, generally have an orientation associated with them, to help in normal calculations. The ordering of the vertices goes *clockwise* around the triangle. Figure below shows what a clockwise ordered triangle would look like.

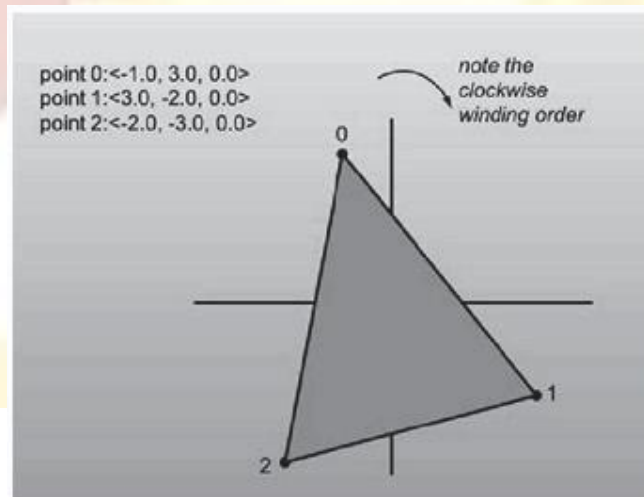


Figure: Three points in space, and the triangle connecting them

When defining a mesh of triangles that define the boundary of a solid, you set it up so that all of the triangles along the skin are ordered clockwise when viewed from the outside.

It is impossible to see triangles that face away from you. (You can find this out by computing the triangle's plane normal and performing a dot product with a vector from the camera location to a location on the plane.)

Now let's move on to the code. To help facilitate using the multiple types, I'll implement triangles structure. I only define constructors and keep the access public.

```
struct tri
{
    type v[3]; // Array access useful for loops

    tri()
    {
        // nothing
    }

    tri( type v0, type v1, type v2 )
    {
        v[0] = v0;
        v[1] = v1;
        v[2] = v2;
    }
};
```

28.3 Strips and Fans

Lists of triangles are generally represented in one of three ways. The first is an explicit list or array of triangles, where every three elements represent a new triangle. However, there are two additional representations, designed to save bandwidth while sending triangles to dedicated hardware to draw them. They are called *triangle strips* and *triangle fans*.

Triangle fans, conceptually, look like the folding fans you see in Asian souvenir shops. They are a list of triangles that all share a common point. The first three elements indicate the first triangle. Then each new element is combined with the first element and the current last element to form a new triangle. Note that an N-sided polygon can be represented efficiently using a triangle fan, Figure below illustrates what I'm talking about.

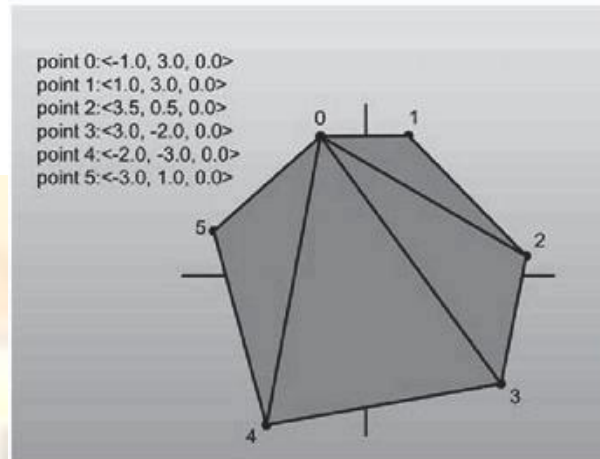


Figure: A list of points composing a triangle fan

Triangles in a triangle strip, instead of sharing a common element with all other triangles like a fan, only share elements with the triangle immediately preceding them. The first three elements define the first triangle. Then each subsequent element is combined with the two elements before it, in clockwise order, to create a new triangle. See Figure below for an explanation of strips.

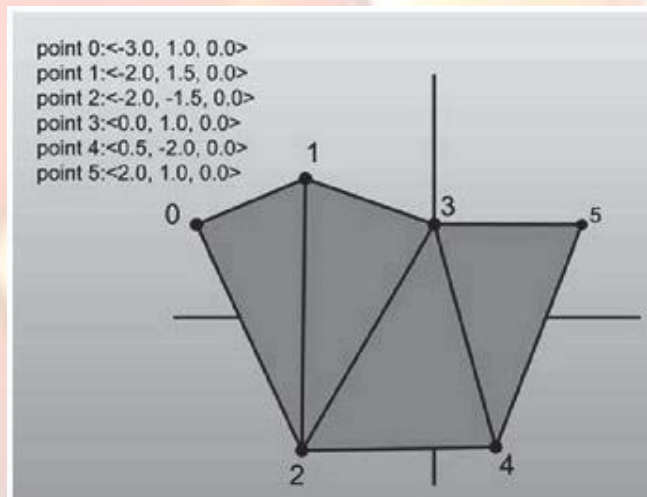


Figure: A list of points composing a triangle strip

28.4 Planes

The next primitive to discuss is the plane. Planes are to 3D what lines are in 2D; they're $n-1$ dimensional hyper-planes that can help you accomplish various tasks. Planes are defined as infinitely large, infinitely thin slices of space, like big pieces of paper. Triangles that make up your model each exist in their own plane. When you have a plane that represents a slice of 3D space, you can perform operations like classification of points and polygons and clipping.

So how do you represent planes? Well it is best to build a structure from the equation that defines a plane in 3D. The implicit equation for a plane is:

$$ax + by + cz + d = 0$$

What do these numbers represent? The triplet $\langle a,b,c \rangle$ represents what is called the normal of the plane. A *normal* is a unit vector that, conceptually speaking, sticks directly out of a plane. A stronger mathematical definition would be that the normal is a vector that is perpendicular to all of the points that lie in the plane.

The d component in the equation represents the distance from the plane to the origin. The distance is computed by tracing a line towards the plane until you hit it. Finally the triplet $\langle x,y,z \rangle$ is any point that satisfies the equation. The set of all points $\langle x,y,z \rangle$ that solve the equation is exactly all the points that lie in the plane.

All of the pictures I'm showing you will be of the top-down variety, and the 3D planes will be on edge, appearing as 2D lines. This makes figure drawing much easier.

Following are two examples of planes. The first has the normal pointing away from the origin, which causes d to be negative (try some sample values for yourself if this doesn't make sense). The second has the normal pointing towards the origin, so d is positive. Of course, if the plane goes through the origin, d is zero (the distance from the plane to the origin is zero). Figures 1 and Figure 2 provide some insight into this relation.

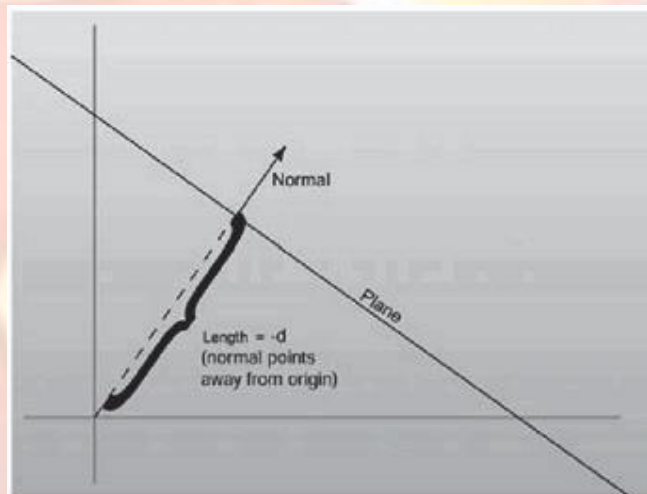


Figure 1: d is negative when the normal faces away from the origin

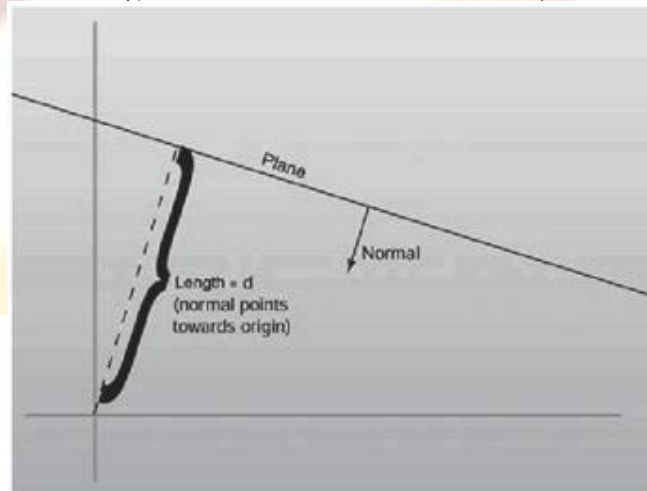


Figure 2: d is positive when it faces towards the origin

It's important to notice that technically the normal $\langle a,b,c \rangle$ does not have to be unit-length for it to have a valid plane equation. But since things end up nicer if the normal is unit-length.

Constructing a plane given three points that lie in the plane is a simple task. You just perform a cross product between the two vectors made up by the three points

$$\langle \text{point}_2 - \text{point}_0 \rangle$$
$$\langle \text{point}_1 - \text{point}_0 \rangle$$

and find a normal for the plane. After generating the normal and making it unit length, finding the d value for the plane is just a matter of storing the negative dot product of the normal with any of the points. This holds because it essentially solves the plane equation above for d . Of course plugging a point in the plane equation will make it equal 0, and this constructor has three of them.

28.5 Back-face Culling

Now that you know how to define a point with respect to a plane, you can perform back-face culling, one of the most fundamental optimization techniques of 3D graphics.

Let's suppose you have a triangle whose elements are ordered in such a fashion that when viewing the triangle from the front, the elements appear in clockwise order. Back-face culling allows you to take triangles defined with this method and use the plane equation to discard triangles that are facing away. Conceptually, any closed mesh, a cube for example, will have some triangles facing you and some facing away. You know for a fact that you'll never be able to see a polygon that faces away from you; they are always hidden by triangles facing towards you. This, of course, doesn't hold if you're allowed to view the cube from its inside, but this shouldn't be allowed to happen if you want to really optimize your engine.

Rather than perform the work necessary to draw all of the triangles on the screen, you can use the plane equation to find out if a triangle is facing towards the camera, and discard it if it is not. How is this achieved? Given the three points of the triangle, you can define a plane that the triangle sits in. Since you know the elements of the triangle are listed in clockwise order, you also know that if you pass the elements in order to the plane constructor, the normal to the plane will be on the front side of the triangle. If you then think of the location of the camera as a point, all you need to do is perform a point-plane test. If the point of the camera is in front of the plane, then the triangle is visible and should be drawn.

There's an optimization to be had. Since you know three points that lie in the plane (the three points of the triangle) you only need to hold onto the normal of the plane, not the entire plane equation. To perform the back-face cull, just subtract one of the triangle's points from the camera location and perform a dot product with the resultant vector and the normal. If the result of the dot product is greater than zero, then the view point was in front of the triangle. Figure below can help explain the point.

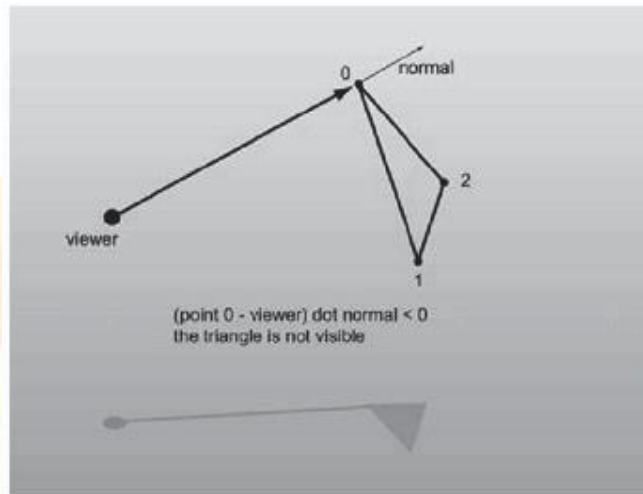


Figure: A visual example of back-face culling

In practice, 3D accelerators can actually perform back-face culling by themselves, so as the triangle rates of cards increase, the amount of manual back-face culling that is performed has steadily decreased. However, the information is useful for custom 3D engines that don't plan on using the facilities of direct hardware acceleration.

28.6 Intersection between a Line and a Plane

This occurs at the point which satisfies both the line and the plane equations.

$$\text{Line equation: } \mathbf{p} = \text{org} + \mathbf{u} * \text{dir} \quad (1)$$

$$\text{Plane equation: } \mathbf{p} * \text{normal} - \mathbf{k} = 0. \quad (2)$$

Substituting (1) into (2) and rearranging we get:

$$(\text{org} + \mathbf{u} * \text{dir}) * \text{normal} - \mathbf{k} = 0$$

$$\text{ie } \mathbf{u} * \text{dir} * \text{normal} = \mathbf{k} - \text{org} * \text{normal}$$

$$\text{ie } \mathbf{u} = (\mathbf{k} - \text{org} * \text{normal}) / (\text{dir} * \text{normal})$$

If $(\text{dir} * \text{normal}) = 0$ then the line runs parallel to the plane and no intersection occurs. The exact point at which intersection does occur can be found by plugging \mathbf{u} back into the line equation in (1).

28.7 Triangle Rasterization

High performance triangle rasterization is a very important topic in Computer Graphics in today's world.

Triangles are the foundation of modern real time graphics, and are by far the most popular rendering primitive. Most computer games released in the last few years are almost completely dependent on triangle rasterization performance. Recently the focus of graphics performance optimization is beginning to shift to bandwidth requirements as well as transformation and lighting. Nevertheless, rasterization performance is still a factor, and this lecture will provide most of the basics of high performance triangle rasterization. Also, it will go into detail about two often neglected rendering quality improvements, sub-pixel and sub-texel accuracy. Also, smooth shading and texture mapping techniques will be described.

28.8 Flat Filling Triangles



Drawing triangle (or in general convex polygon, but as we discussed we will use only triangles) is very simple. The basic idea of the line triangle drawing algorithm is as follows.

For each scan line (horizontal line on the screen), find the points of intersection with the edges of the triangle. Then, draw a horizontal line between intersections and do this for all scan lines.

But how can we find these points quickly?
Using linear interpolation!

We have 3 vertices and we want to find coordinates of all points belonging to segments determined by these vertices.

Assume we have segment given by points:
(x_a, y_a) and (x_b, y_b).

Our task is to find points: (x_c, y_a+1), (x_d, y_a+2), ..., (x_m, y_b-1), (x_n, y_b).

Notice that x_a changes to x_b in ($y_b - y_a$) steps.

We also have:

$$x_a = x_a + 0 * (x_b - x_a) / (y_b - y_a),$$

$$x_b = x_a + (y_b - y_a) * (x_b - x_a) / (y_b - y_a)$$

$$\text{and, in general, } x_i = x_a + (y_i - y_a) * \text{delta},$$

$$\text{where } \text{delta} = (x_b - x_a) / (y_b - y_a).$$

The general function for linear interpolation is:

$$f(X) = A + X * ((B - A) / \text{steps}) \text{ where we slide from } A \text{ to } B \text{ in } \text{steps} \text{ steps}$$

Here is pseudo code for a triangle filling algorithm.

- The coordinates of vertices are (A_x, A_y), (B_x, B_y), (C_x, C_y); we assume that $A_y \leq B_y \leq C_y$ (you should sort them first)
- dx_1, dx_2, dx_3 are deltas used in interpolation
- Horizontal line draws horizontal segment with coordinates (S_x, Y), (E_x, Y)
- S_x, E_x are left and right x-coordinates of the segment we have to draw
- $S = A$ means that $S_x = A_x$; $S_y = A_y$;

```
if (B.y - A.y > 0) dx1 = (B.x-A.x) / (B.y-A.y)
else dx1=0;
```

```
if (C.y - A.y > 0) dx2=(C.x-A.x)/(C.y-A.y)
else dx2=0;
```

```
if (C.y-B.y > 0) dx3=(C.x-B.x)/(C.y-B.y)
else dx3=0;
```

```
S = E = A
```

```
if(dx1 > dx2)
{
    for(; S.y<=B.y; S.y++, E.y++, S.x+=dx2, E.x += dx1)
        horizontal line (S.x,E.x,S.y, E.x,color);

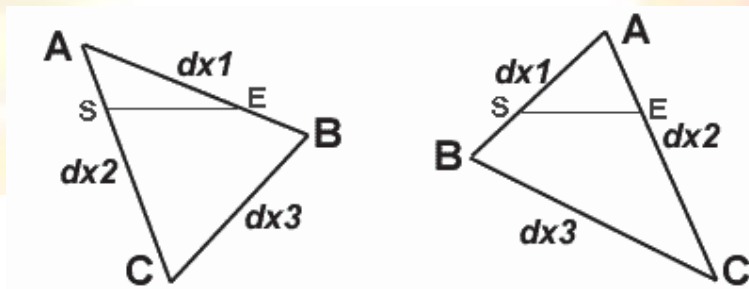
    E=B;

    for(;S.y<=C.y;S.y++,E.y++,S.x+=dx2,E.x+=dx3)
        horizontal line(S.x,E.x,S.y, E.x,color);
}
else
{
    for(;S.y<=B.y;S.y++,E.y++,S.x+=dx1,E.x+=dx2)
        horizontal line(S.x,E.x,S.y, E.x,color);

    S=B;

    for(;S.y<=C.y;S.y++,E.y++,S.x+=dx3,E.x+=dx2)
        horizontal line(S.x,E.x,S.y, E.x,color);
}
```

I ought to explain what is the comparison $dx1 > dx2$ for. It's optimization trick: in the horizontal line routine, we don't need to compare the x's (S.x is always less than or equal to E.x).



28.9 Gouraud Shading



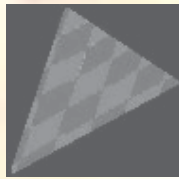
The idea of gouraud and flat triangle is nearly the same. Gouraud takes only three parameters more (the color value of each of the vertices), and the routine just interpolates among them drawing a beautiful, shaded triangle.

You can use 256-colors mode, in which vertices' colors are simply indices to palette or hi-color mode (recommended).

Flat triangle interpolated only one value (x in connection with y), 256 colors gouraud needs three (x related to y , color related to y , and color related to x), hi-color gouraud needs seven (x related to y , red, green and blue components of color related to y , and color related to x (also three components))

Drawing a gouraud triangle, we add only two parts to the flat triangle routine. The horzline routine gets a bit more complicated due to the interpolation of the color value related to x but the main routine itself remains nearly the same.

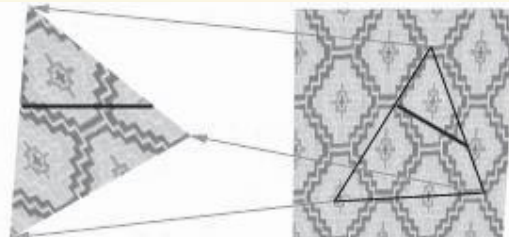
28.10 Textured Triangles



We can also apply any bitmap on triangle for filling it.

I'll show you the idea of linear (or 'classical') texture mapping (without perspective correction). Linear mapping works pretty well (read: fast) in some scenes, but perspective correction is in some way needed in most 3D systems.

Again we're using the idea of interpolation: now we'll code a texture triangle filler. And again the idea is perfectly the same, only two more values to interpolate, that is five values total. In texture mapping, we interpolate x , u , and v related to y , and u and v related to x (u and v are coordinates in the 2D bitmap space). The situation is maybe easier to understand by looking at the following picture:



The left triangle is the triangle which is drawn onto the screen. There's a single scanline (one call to the horizline routine) pointed out as an example. The triangle on the right is the same triangle in the bitmap space, and there's the same scanline drawn from another point of view into it, too. So we need just to interpolate, interpolate, and once more interpolate in texture filler - an easy job if you've understood the idea of gouraud filler.

28.11 COLOR

It is important to understand how color is represented in computer graphics so that we can manipulate it effectively. A color is usually represented in the graphics pipeline by a three-element vector representing the intensities of the red, green, and blue components, or for a more complex object, by a four-element vector containing an additional value called the *alpha* component that represents the opacity of the color. Thus we can talk about **rgb** or **rgba** colors and mean a color that's made up of either three or four elements. There are many different ways of representing the intensity of a particular color element. *Colors can also be represented as floating point values in the range [0,1].*

Nowadays every PC we can buy has hardware that can render images with thousands or millions of individual colors. Rather than have an array with thousands of color entries, the images instead contain explicit color values for each pixel. A 16-bit display is named since each pixel in a 16-bit image is taken up by 16 bits (2 bytes): 5 bits of red information, 6 bits of green information, and 5 bits of blue information. Incidentally, the extra bit (and therefore twice as much color resolution) is given to green because our eyes are more sensitive to green. A 24-bit display, of course, uses 24 bits, or 3 bytes per pixel, for color information. This gives 1 byte, or 256 distinct values each, for red, green, and blue. This is generally called *true color*, because 256^3 (16.7 million) colors is about as much as your eyes can discern, so more color resolution really isn't necessary, at least for computer monitors.

Finally, there is 32-bit color, something seen on most new graphics cards. Many 3D accelerators keep 8 extra bits per pixel around to store transparency information, which is generally referred to as the *alpha channel*, and therefore take up 4 bytes, or 32 bits, of storage per pixel. Rather than reimplement the display logic on 2D displays that don't need alpha information, these 8 bits are usually just wasted.

WHY WE MIGHT WANT 128-BIT COLOR?

In one of the early magazines articles of *Mike Abrash* [ABRASH 1992], he tells a story about going from a 256-color palette to hardware that supported 256 levels for each RGB color—16 million colors! What would we do with all those colors? He goes on to tell of a story by Sheldon Linker at the eighth Annual Computer Graphics Show on how the folks at the Jet Propulsion Lab back in the 1970s had a printer that could print over 50 million distinct colors. As a test, they printed out words on paper where the background color was only one color index from the word's color. To their surprise, it was easy to discern the words—the human eye is very sensitive to color graduations and edge detection. The JPL team then did the same tests on color monitors and discovered that only about 16 million colors could be distinguished. It seems that the eye is (not too surprisingly) better at perceiving detail from reflected light (such as from a printed page) than from emissive light (such as from a CRT). The moral is that the eye is a lot more perceptive than you might think. Twenty four-bits of color really is not that much range, particularly if we are performing multiple passes. Round-off error can and will show up if we aren't careful!

An example of the various gamuts is shown in the figure below. The CIE diagrams are the traditional way of displaying perceived color space, which, we should note, is very different from the linear color space used by today's graphics hardware. The colored area is the gamut of the human eye. The gamut of printers and monitors are subsets of this gamut.

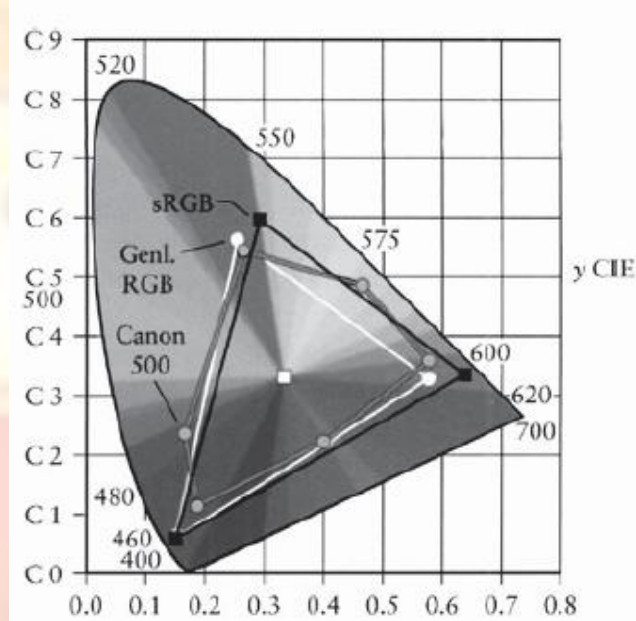


Figure 1: The 1931 CIE diagram shows the gamut of the eye and the lesser gamut of output devices.

28.12 Multiplying Color Values

First we need to be aware of how to treat colors. The calculation of the color of a particular pixel depends, for example, on the surface's material properties that we've programmed in, the color of the ambient light (*lighting model*), the color of any light shining on the surface (perhaps of the angle of the light to the surface), the angle of the surface to the viewpoint, the color of any fog or other scattering material that's between the surface and the viewpoint, etc. No matter how you are calculating the color of the pixel, it all comes down to color calculations, at least on current hardware, on **rgb** or **rgba** vectors where the individual color elements are limited to the [0,1] range. Operations on colors are done piecewise—that is, even though we represent colors as **rgb** vectors, they aren't really vectors in the mathematical sense. Vector multiplication is different from the operation we perform to multiply color. We'll use the \otimes symbol to indicate such piecewise multiplication.

Colors are multiplied to describe the interaction between a surface and a light source. The colors of each are multiplied together to estimate the reflected light color—this is the color of the light that this particular light reflects off this surface. The problem with the standard **rgb** model is just that we're simulating the entire visible spectrum by three colors with a limited range.

Let's start with a simple example of using reflected colors. Later on we will discuss on lighting, we'll discover how to calculate the intensity of a light source, but for now, just assume that we've calculated the intensity of a light, and it's a value called i_d . This intensity of our light is represented by, say, a nice lime green color.

Thus

$$\text{light color } i_d = [0.34765, 0.92578, 0.24609]$$

Let's say we shine this light on a nice magenta surface given by c_s .

$$\text{surface color } c_s = [0.86719, 0.00000, 0.98828]$$

So, to calculate the color contribution of this surface from this particular light, we perform a piecewise multiplication of the color values.

$$\begin{aligned} i_d \otimes c_s &= [0.34765, 0.92578, 0.24609] \otimes [0.86719, 0.00000, 0.98828] \\ &= [(0.34765)(0.86719), (0.92578)(0), (0.24609)(0.98828)] \\ &= [0.30148, 0.00000, 0.243210] \end{aligned}$$

Note: Piecewise multiplication is denoted by $i_d \otimes c_s$ that is element-by-element multiplication. Used in color operations, where the vector just represents a convenient notation for an array of scalars that are operated on simultaneously but independently.

This gives us the dark plum color shown in figure below. We should note that since the surface has no green component, that no matter what value we used for the light color, there would *never* be any green component from the resulting calculation. Thus a pure green light would provide no contribution to the intensity of a surface if that surface contained a zero value for its green intensity. Thus it's possible to illuminate a surface with a bright light and get little or no illumination from that light. We should also note that using anything other than a full-bright white light $[1,1,1]$ will involve multiplication of values less than one, which means that using a single light source will only illuminate a surface to a maximum intensity of its color value, never more. This same problem also happens when a texture is modulated by a surface color. The color of the surface will be multiplied by the colors in the texture. If the surface color is anything other than full white, the texture will become darker. Multiple texture passes can make a surface very dark very quickly.

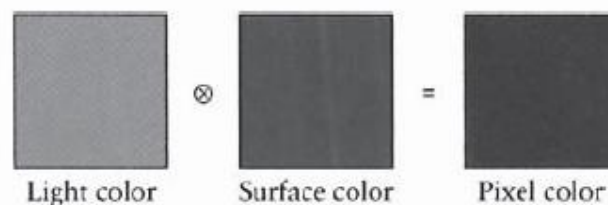


Figure 2: Multiplying (modulating) color values results in a color equal to or less than (darker) the original two.

Given that using a colored light in a scene makes the scene darker, how do we make the scene brighter? There are a few ways of doing this. Given that color multiplication will never result in a brighter color, it's offset a bit since we end up summing all the light contributions together, which, as we'll see in the next section, brings with it its own problems. But if we are just interested in increasing the brightness on one particular light or texture, one way is to use the API (*Library routines e.g. OpenGL or DirectX*) to artificially brighten the source—this is typically done with texture preprocessing. Or, we can artificially brighten the source, be it a light or a texture, by adjusting the values after we modulate them.

28.13 Dealing with Saturated Colors

On the other hand, what if we have *too* much contribution to a color? While the colors of lights are modulated by the color of the surface, *each* light source that illuminates the surface is added to the final color. All these colors are summed up to calculate the final color. Let's look at such a problem. We'll start with summing the reflected colors off a surface from two lights. The first light is an orange color and has *rgb* values [1.0,0.49,0.0], and the second light is a nice light green with **rgb** values [0.0,1.0,0.49]. Summing these two colors yields [1.0, 1.49, 0.49], which we can't display because of the values larger than one figure below shows.

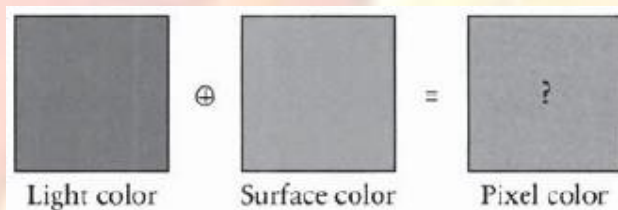


Figure 3: Adding colors can result in colors that are outside the displayable range.

So, what can be done when color values exceed the range that the hardware can display? It turns out that there are three common approaches [HALL 1990].

Clamping the color values is implemented in hardware, so for shaders (*technology used in today computer graphics for lighting and shading*), it's the default, and it just means that we clamp any values outside the [0,1] range. Unfortunately, this results in a shift in the color.

The second most common approach is to scale the colors by the largest component. This maintains the color but reduces the overall intensity of the color.

The third is to try to maintain the intensity of the color by shifting (or clipping) the color toward pure bright white by reducing the colors that are too bright while increasing the other colors and maintaining the overall intensity. Since we can't see what the actual color for (figure above) is, let's see what color each of these methods yields (figure below).

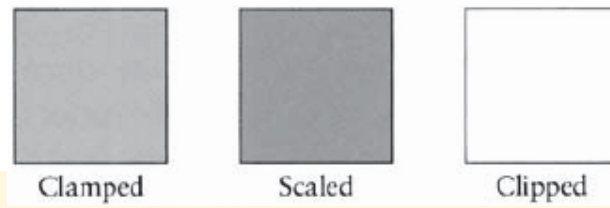


Figure 4: The results of three strategies for dealing with the same oversaturated color.

As we can see, we get three very different results. In terms of perceived color, the scaled is probably the closest though it's darker than the actual color values. If we weren't interested in the color but more in terms of saturation, then the clipped color is closer. Finally, the clamped value is what we get by default, and as you can see, the green component is biased down so that we lose a good sense of the "greenness" of the color we were trying to create.



Lecture No.29 Mathematics of Lighting and Shading Part III

Traditional 3D Hardware-Accelerated Lighting Models

We will now take a look at the traditional method of calculating lighting in hardware—a method that we'll find is sufficient for most of our needs. The traditional approach in real-time computer graphics has been to calculate lighting at a vertex as a sum of the ambient, diffuse, and specular light. In the simplest form (used by OpenGL and Direct3D), the function is simply the sum of these lighting components (clamped to a maximum color value). Thus we have an ambient term and then a sum of all the light from the light sources.

$$i_{total} = k_a i_a + \sum (k_d i_d + k_s i_s)$$

Where i_{total} , is the intensity of light (as an **rgb** value) from the sum of the intensity of the global ambient value and the diffuse and specular components of the light from the light sources. This is called a *local lighting model* since the only light on a vertex is from a light source, not from other objects. That is, lights are lights, not objects. Objects that are brightly lit don't illuminate or shadow any other objects. We've included the *reflection coefficients* for each term, **k** for completeness since we'll frequently see the lighting equation. The reflection coefficients are in the [0, 1] range and are specified as part of the material property. However, they are strictly empirical and since they simply adjust the overall intensity of the material color, the material color values are usually adjusted so the color intensity varies rather than using a reflection coefficient, so we'll ignore them in our actual color calculations. This is a very simple lighting equation and gives fairly good results. However, it does fail to take into account any gross roughness or anything other than perfect isotropic reflection. That is, the surface is treated as being perfectly smooth and equally reflective in all directions. Thus this equation is really only good at modeling the illumination of objects that don't have any "interesting" surface properties. By this we mean anything other than a smooth surface (like fur or sand) or a surface that doesn't really reflect light uniformly in all directions (like brushed metal, hair, or skin). However, with liberal use of texture maps to add detail, this model has served pretty well and can still be used for a majority of the lighting processing to create a realistic environment in real time. Let's take a look at the individual parts of the traditional lighting pipeline.

Ambient Light

Ambient light is the light that comes from all directions—thus all surfaces are illuminated equally regardless of orientation. However, this is a big hack in traditional lighting calculations since "real" ambient light really comes from the light reflected from the "environment." This would take a long time to calculate and would require ray tracing or the use of radiosity methods, so traditionally, we just say that there's x amount of global ambient light and leave it at that. This makes ambient light a little different from the other lighting components since it doesn't depend on a light source. However, we typically *do* want ambient light in our scene because having a certain amount of ambient light makes the scene look natural. One large problem with the simplified lighting model is that there is no illumination of an object with reflected light—the calculations required are enormous for a scene of any complexity (every object can potentially reflect some light and provide some illumination for every other object in a scene) and are too time consuming to be considered for real-time graphics. So, like most things in computer

graphics, we take a look at the real world, decide it's too complicated, and fudge up something that kind of works. Thus the ambient light term is the "fudge factor" that accounts for our simple lighting model's lack of an inter-object reflectance term. The ambient light equation is given by

$$i_a = m_a \cdot s_a$$

Where i_a is the ambient light intensity, m_a is the ambient material color, and s_a is the light source ambient color. Typically, the ambient light is some amount of white (i.e., equal **rgb** values) light, but we can achieve some nice effects using colored ambient light. Though it's very useful in a scene, ambient light doesn't help differentiate objects in a scene since objects rendered with the same value of ambient tend to blend since the resulting color is the same. Figure 1 shows a scene with just ambient illumination. We can see that it's difficult to make out details or depth information with just ambient light.



Figure 1: Ambient light provides illumination, but no surface details.

Ambient lighting is our friend. With it we make our scene seem more realistic than it is. A world without ambient light is one filled with sharp edges, of bright objects surrounded by sharp, dark, harsh shadows. A world with too much ambient light looks washed out and dull. Since the number of actual light sources supported by hardware FFP is limited (typically to eight simultaneous), we'll be better off to apply the lights to add detail to the area that our user is focused on and let ambient light fill in the rest. Before we point out that talking about the hardware limitation of the number of lights has no meaning on shaders, where *we* do the lighting calculations, we'll point out that eight lights were typically the maximum that the hardware engineers created for *their* hardware. It was a performance consideration. There's nothing stopping us (except buffer size) from writing a shader that calculates the effects from a hundred simultaneous lights. But we think that we'll find that it runs much too slowly to be used to render our entire scene. But the nice thing about shaders is we *can*.

Diffuse Light

Diffuse light is the light that is absorbed by a surface and is reflected in all directions. In the traditional model, this is *ideal* diffuse reflection—good for rough surfaces where the reflected intensity is constant across the surface and is independent of viewpoint but

depends only upon the direction of the light source to the surface. This means that regardless of the direction from which we view an object with a stationary diffuse light source on it, the brightness of any point on the surface will remain the same. Thus, unlike ambient light, the intensity of diffuse light is directional and is a function of the angle of the incoming light and the surface. This type of shading is called *Lambertian shading* after Lambert's cosine law, which states that the intensity of the light reflected from an ideal diffuse surface is proportional to the cosine of the direction of the light to the vertex normal. Since we're dealing with vertices here and not surfaces, each vertex has a normal associated with it. We might hear talk of per-vertex normals vs. per-polygon normals. The difference being that per polygon has one normal shared for all vertices in a polygon, whereas per vertex has a normal for each vertex. OpenGL has the ability to specify per-polygon normals, and Direct3D does not. Since vertex shaders can't share information between vertices (unless we explicitly copy the data our self). We'll focus on per-vertex lighting. Figure 2 shows the intensity of reflected light as a function of the angle between the vertex normal and the light direction.

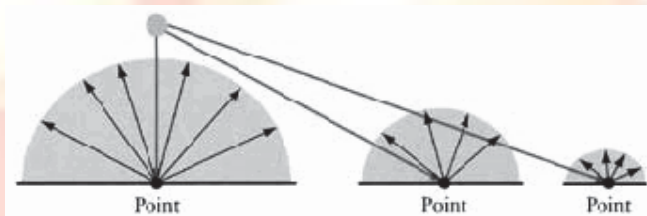


Figure 2: Diffuse light decreases as the angle between the light vector and the surface normal increases.

The equation for calculating diffuse lighting is

$$i_d = (\hat{n} \cdot \hat{l})(m_d \otimes s_d)$$

Which is similar to the ambient light equation, except that the diffuse light term is now multiplied by the dot product of the unit normal of the vertex and the unit direction vector *to* the light from the vertex (not the direction *from* the light). Note that the **md** value is a color vector, so there are **rgb** or **rgba** values that will get modulated.

Since $(\hat{n} \cdot \hat{l}) = |\hat{n}||\hat{l}| \cos(\theta)$, where theta is the angle between vectors, when the angle between them is zero, $\cos(\theta)$ is 1 and the diffuse light is at its maximum. When the angle is 90° , $\cos(\theta)$ is zero and the diffuse light is zero. One calculation advantage is that when the $\cos(\theta)$ value is negative, this means that the light isn't illuminating the vertex at all. However, since we (probably!) don't want the light illuminating sides that it physically can't shine on, we want to clamp the contribution of the diffuse light to contribute only when $\cos(\theta)$ is positive. Thus the equation in practice looks more like

$$i_d = \text{MAX}(0, (\hat{n} \cdot \hat{l}))(m_d \otimes s_d)$$

Where we've clamped the diffuse value to only positive values, Figure 3 was rendered with just diffuse lighting. Notice how we can tell a lot more detail about the objects and pick up distance cues from the shading.



Figure 3: Diffuse shading brings out some surface details.

The problem with just diffuse lighting is that it's independent of the viewer's direction. That is, it's strictly a function of the surface normal and the light direction. Thus as we change the viewing angle to a vertex, the vertex's diffuse light value never changes. You have to rotate the object (change the normal direction) or move the light (change the light direction) to get a change in the diffuse lighting of the object. However, when we combine the ambient and diffuse, as in Figure 4, we can see that the two types of light give a much more realistic representation than either does alone. This combination of ambient and diffuse is used for a surprisingly large number of items in rendered scenes since when combined with texture maps to give detail to a surface we get a very convincing shading effect.



Figure 4: When diffuse and ambient terms are combined, you get more detail and a more natural-looking scene. The final color is the combination of the ambient and diffuse colors.

Specular Light

Ambient light is the light that comes from the environment (i.e., it's directionless); diffuse light is the light from a light source that is reflected by a surface evenly in all directions (i.e., it's independent of the viewer's position). Specular light is the light from a light source that is reflected by a surface and is reflected in such a manner that it's both a function of the light's vector and the viewer's direction. While ambient light gives the

object an illuminated matte surface, specular light is what gives the highlights to an object. These highlights are greatest when the viewer is looking directly along the reflection angle from the surface. This is illustrated in Figure 5.

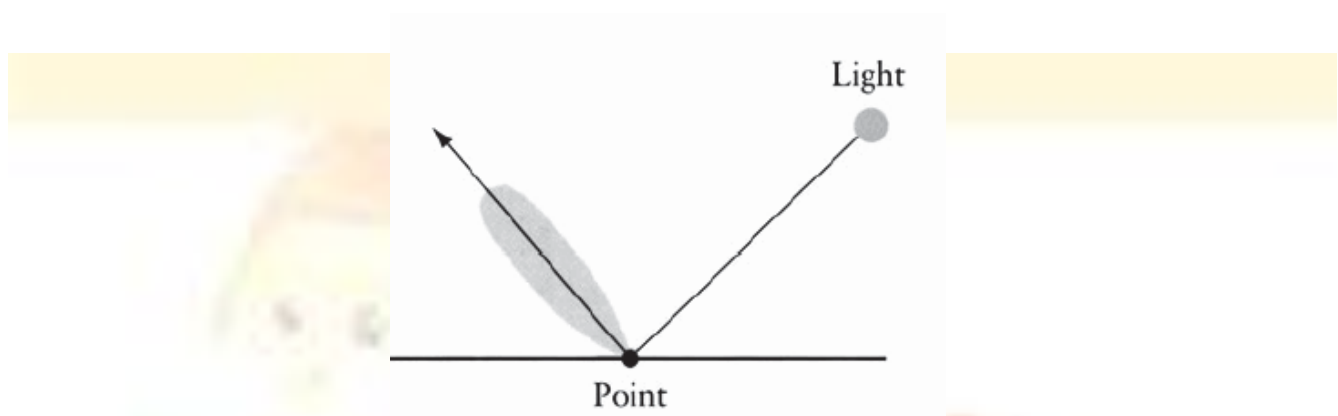


Figure 5: Specular light's intensity follows the reflection vector.

Most discussions of lighting (including this one) start with Phong's lighting equation (which is not the same as Phong's shading equation). In order to start discussing specular lighting, let's look at a diagram of the various vectors that are used in a lighting equation. We have a light source, some point the light is shining on, and a viewpoint. The light direction (from the point *to* the light) is vector l , the reflection vector of the light vector (as if the surface were a mirror) is r , the direction *to* the viewpoint from the point is vector v . The point's normal is n .

Phong's Specular Light Equation

Warnock [WARNOCK 1969] and Romney [ROMNEY 1969] were the first to try to simulate highlights using a $\cos^n(\theta)$ term. But it wasn't until Phong Bui-Tong [BUI 1998] reformulated this into a more general model that formalized the power value as a measure of surface roughness that we approach the terms used today for specular highlights. Phong's equation for specular lighting is

$$i_s = (m_s \otimes s_s)(\hat{r} \cdot \hat{v})^{m_s} \quad (\text{Phong})$$

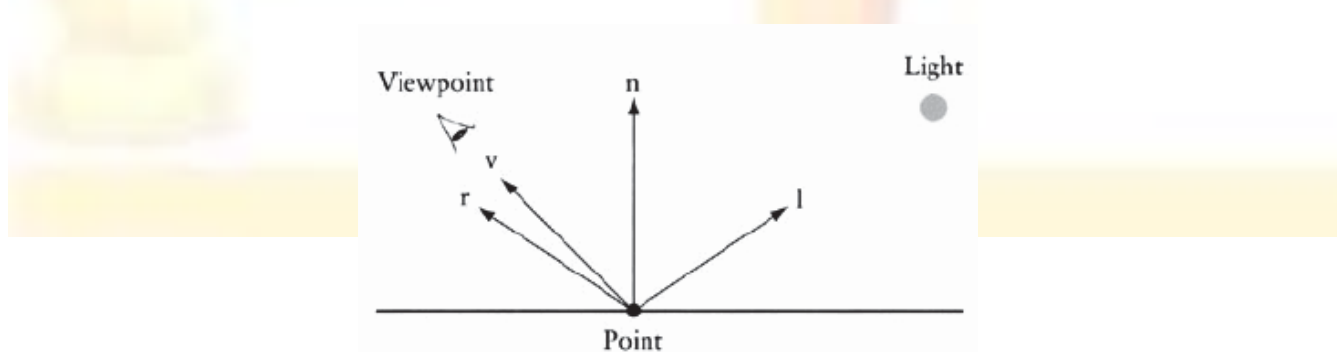


Figure 6: The relationship between the normal n , the light vector l , the view direction v , and the reflection vector r .

It basically says that the more the view direction, \mathbf{v} , is aligned with the reflection direction, \mathbf{r} , the brighter the specular light will be. The big difference is the introduction of the \mathbf{ms} term, which is a power term that attempts to approximate the distribution of specular light reflection. The \mathbf{ms} term is typically called the "shininess" value. The larger the \mathbf{ms} value, the "tighter" (but not brighter) the specular highlights will be. This can be seen in the Figure 7, which shows values of $(\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^m$ for values of \mathbf{m} ranging from 1 to 128. As we can see, the specular highlights get narrower for higher values, but they don't get any brighter.

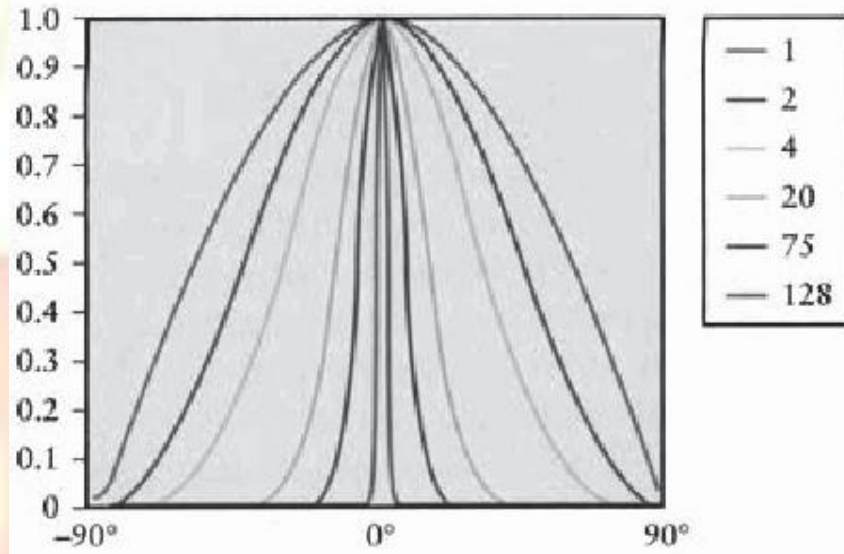


Figure 7: Phong's specular term for various values of the "shininess" term. Note that the values never get above 1.

Now, as we can see, this requires some calculations since we can't know \mathbf{r} before hand since it's the \mathbf{v} vector reflected around the point's normal. To calculate \mathbf{r} we can use the following equation:

$$\mathbf{r} = \frac{2(\mathbf{n} \cdot \mathbf{l})\mathbf{n} - \mathbf{l}}{|\mathbf{n}|^2}$$

If \mathbf{l} and \mathbf{n} are normalized, then the resulting \mathbf{r} is normalized and the equation can be simplified.

$$\hat{\mathbf{r}} = 2(\hat{\mathbf{n}} \cdot \hat{\mathbf{l}})\hat{\mathbf{n}} - \hat{\mathbf{l}}$$

And just as we did for diffuse lighting, if the dot product is negative, then the term is ignored.

$$i_s = \text{MAX}\left(0, (\hat{\mathbf{r}} \cdot \hat{\mathbf{v}})^m, (m_s \otimes s_s)\right) \quad (\text{Phong})$$

Figure 8 shows the scene with just specular lighting. As we can see, we get an impression of a very shiny surface.

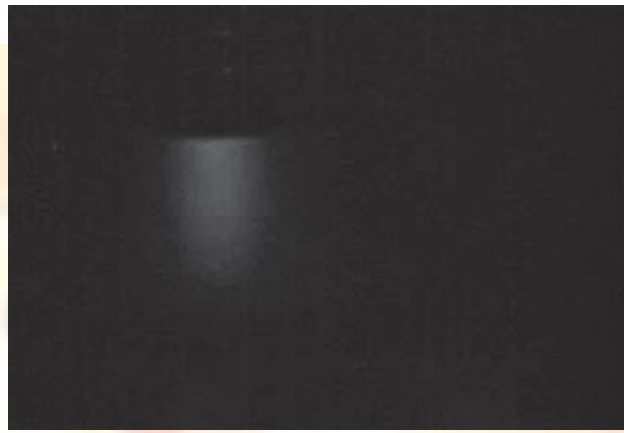


Figure 8: A specular term just shows the highlights.

When we add the ambient, diffuse, and specular terms together, we get Figure 8A. The three terms all act in concert to give us a fairly good imitation of a nice smooth surface that can have a varying degree of shininess to it. We may have noticed that computing the reflection vector took a fair amount of effort. In the early days of computer graphics, there was a concerted effort to reduce anything that took a lot of computation, and the reflection vector of Phong's equation was one such item.



Figure 8A: A combination of ambient, diffuse, and specular illumination.

Blinn's Simplification: OpenGL and DirectX Lighting

Now it's computationally expensive to calculate specular lighting using Phong's equation since computing the reflection vector is expensive. Blinn [BLINN 1977] suggested, instead of using the reflection and view vectors, that we create a "half" vector that lies between the light and view vectors. This is shown as the \mathbf{h} vector in Figure 9. Just as Phong's equation maximizes when the reflection vector is coincident with the view vector (thus the viewer is looking directly along the reflection vector), so does Blinn's. When the half vector is coincident with the normal vector, then the angle between the view vector

and the normal vector is the same as between the light vector and the normal vector. Blinn's version of Phong's equation is:

$$i_s = (m_s \otimes s_s)(\hat{n} \cdot \hat{h})^{m_s} \quad (\text{Blinn-Phong})$$

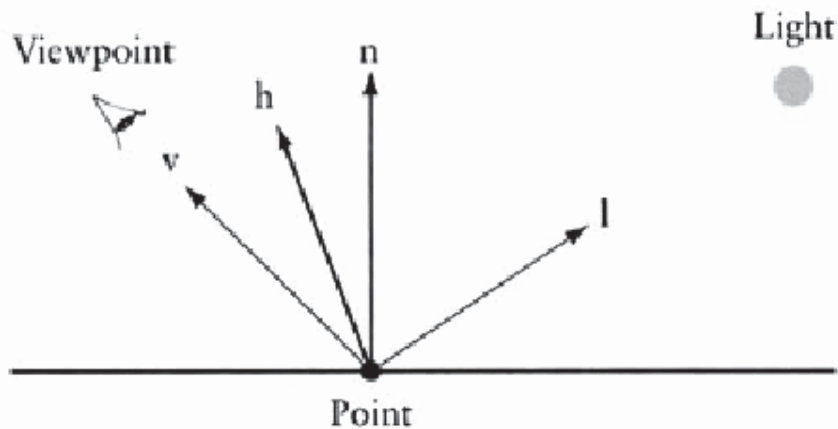


Figure 9: The half-angle vector is an averaging of the light and view vectors. where the half vector is defined as

$$h = \frac{l + v}{|l + v|}$$

The advantage is that no reflection vector is needed; instead, we can use values that are readily available, namely, the view and light vectors. Note that both OpenGL and the DirectX FFP use Blinn's equation for specular light. Besides a speed advantage, there are some other effects to note between Phong's specular equation and Blinn's. If we multiply Blinn's exponent by 4, we approximate the results of Phong's equation. Thus if there's an upper limit on the value of the exponent, Phong's equation can produce sharper highlights. For $l \cdot v$ angles greater than 45° (i.e., when the light is behind an object and we're looking at an edge), the highlights are longer along the edge direction for Phong's equation. Blinn's equation produces results closer to those seen in nature.

For an in-depth discussion of the differences between the two equations, there's an excellent discussion in [FISHER 1994]. Figure 10 shows the difference between Phong lighting and Blinn—Phong lighting.

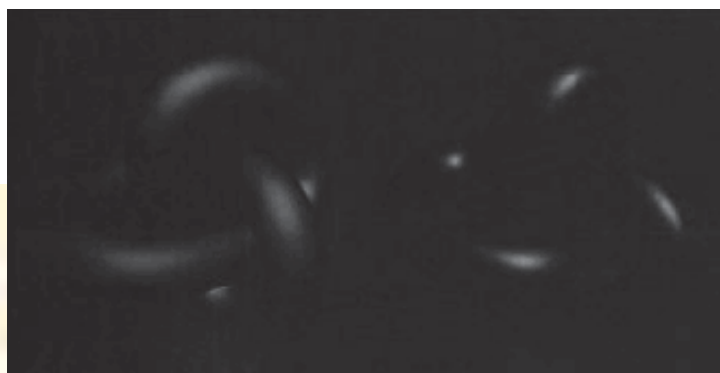


Figure 10: Blinn-Phong specular on the left, Phong specular on the right.

Note: Some of the material, for the preparation of this lecture, is taken from a book Real time shader Programming by Ron fosner

Lecture No.30 Mathematics of Lighting and Shading Part IV

The Lighting Equation

So now that we've computed the various light contributions to our final color value, we can add them up to get the final color value. Note that the final color values will have to be made to fit in the [0,1] range for the final **rgb** values.

$$i_{\text{total}} = i_a + \sum (i_d + i_s)$$

Our final scene with ambient, diffuse, and (Blinn's) specular light contributions (with one white light above and to the left of the viewer) looks like Figure 1.



Figure 1: A combination of ambient, diffuse, and specular illumination.

It may be surprising to discover that there's more than one way to calculate the shading of an object, but that's because the model is empirical, and there's no correct way, just different ways that all have tradeoffs. Until now though, the only lighting equation we've been able to use has been the one we just formulated. Most of the interesting work in computer graphics is tweaking that equation, or in some cases, throwing it out altogether and coming up with something new.

The next sections will discuss some refinements and alternative ways of calculating the various coefficients of the lighting equation.

Light Attenuation

Light in the real world loses its intensity as the inverse square of the distance from the light source to the surface being illuminated. However, when put into practice, this seemed to drop off the light intensity in too abrupt a manner and then not to vary too much after the light was far away. An empirical model was developed that seems to give satisfactory results. This is the attenuation model that's used in OpenGL and DirectX. The *fatten* factor is the attenuation factor. The distance *d* between the light and the vertex is always positive. The attenuation factor is calculated by the following equation:

$$f_{\text{atten}} = 1 / (k_c + k_l d + k_q d^2)$$

Where the k_c , k_l , and k_q parameters are the constant, linear, and quadratic attenuation constants respectively, to get the "real" attenuation factor, we can set k_q to one and the others to zero. The attenuation factor is multiplied by the light diffuse and specular values. Typically, each light will have a set of these parameters for itself. The lighting equation with the attenuation factor looks like this.

$$i_{\text{total}} = i_a + \sum f_{\text{atten}}(i_d + i_s)$$

Figure 2 shows a sample of what attenuation looks like. This image is the same as the one shown in Figure 1, but with light attenuation added.



Figure 2: A scene with light attenuation. The white sphere is the light position.

Schlick's simplification for the Specular Exponential Term

Real-time graphics programmers are always looking for simplifications. We've probably gathered that there's no such thing as the "correct" lighting equation, just a series of hacks to make things look right with as little computational effort as possible. Schlick [SCHLICK 1994] suggested a replacement for the exponential term since that's a fairly expensive operation. If we define part of our specular light term as follows:

$$(S)^{m_s}$$

where S is either the Phong or Blinn-Phong flavor of the specular lighting equation, then Schlick's simplification is to replace the preceding part of the specular equation with

$$\frac{S}{m_s - m_s S + S}$$

Which eliminates the need for an exponential term, At first glance, a plot of Schlick's function looks very similar to the exponential equation (Figure 3).

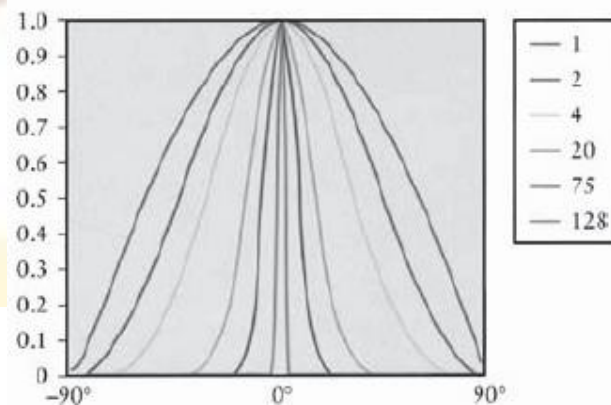


Figure 3: Schlick's term for specular looks very much like the more expensive Phong term.

If we plot both equations in the same graph (Figure 4), we can see some differences and evaluate just how well Schlick's simplification works. The blue values are Schlick's, and the red are the exponential plot. As the view and light angles get closer (i.e., get closer to zero on the x axis), we can see that the values of the curves are quite close. (For a value of zero, they overlap.) As the angles approach a grazing angle, we can see that the approximation gets worse. This would mean that when there is little influence from a specular light, Schlick's equation would be slightly less sharp for the highlight.

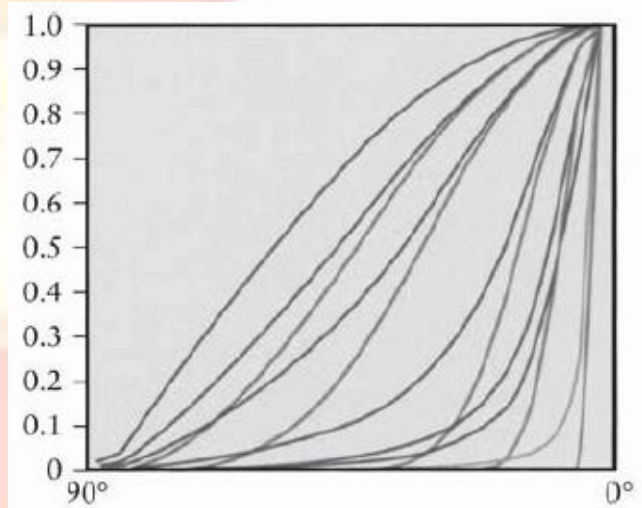


Figure 4: Schlick's vs. Phong's specular terms.

We might notice the green line in Figure 4. Unlike the limit of a value of 128 for the exponential imposed in both OpenGL and DirectX FFP, we can easily make our values in the approximation any value we want. The green line is a value of 1024 in Schlick's equation. We may be thinking that we can make a very sharp specular highlight using Schlick's approximation with very large values—sharper than is possible using the exponential term. Unfortunately, we can't since we really need impractically large values (say, around 100 million) to boost it significantly over the exponential value for 128. But that's just the kind of thinking that's going to get our creative juices flowing when writing our own shaders! If the traditional way doesn't work, figure out something that will.

Oren—Nayar Diffuse Reflection

Though there's been a lot of research on specular reflection models, there's been less research on diffuse reflection models. One of the problems of the standard Lambertian model is that it considers the surface as a smooth diffuse surface. Surfaces that are really rough, like sandpaper, exhibit much more of a backscattering effect, particularly when the light source and the view direction are in the same direction.

The classic example of this is a full moon. If we look at the picture of the moon shown in Figure 5, it's pretty obvious that this doesn't follow the Lambertian distribution—if it did, the edges of the moon would be in near darkness. In fact, the edges look as bright as the center of the moon. This is because the moon's surface is rough—the surface is made of a jumble of dust and rock with diffuse reflecting surfaces at all angles—thus the quantity of reflecting surfaces is uniform no matter the orientation of the surface; hence no matter the orientation of the surface to the viewer, the amount of light reflecting off the surface is nearly the same.



Figure 5: The full moon is a good example of something that doesn't show Lambertian diffuse shading.

The effect we're looking at is called backscattering. Backscattering is when a rough surface bounces around a light ray and then reflects the ray in the direction the light originally came from. Note that there is a similar but different effect called retro reflection. Retro reflection is the effect of reflecting light toward the direction from which it came, no matter the orientation of the surface. This is the same effect that we see on bicycle reflectors. However, this is due to the design of the surface features (made up of vshaped or spherical reflectors) rather than a scattering effect.

In a similar manner, when the light direction is closer to the view direction, we get the effect of forward scattering. Forward scattering is just backscattering from a different direction. In this case, instead of near uniform illumination though, we get near uniform loss of diffuse lighting. We can get the same effects here on Earth. Figures 6 and 7, shows the same surfaces demonstrating backscattering and forward scattering. Both the dirt field in Figure 6 and the soybean field in Figure 7 can be considered rough diffuse reflecting surfaces.



Figure 6: The same dirt field showing wildly differing reflection properties.



Figure 7: A soybean field showing differing reflection properties.

Notice how the backscattering image shows a near uniform diffuse illumination, whereas the forward scattering image shows a uniform dull diffuse illumination. Also note that we can see specular highlights and more color variation because of the shadows due to the rough surface whereas the backscattered image washes out the detail. In an effort to better model rough surfaces, Oren and Nayar [OREN 1992] came up with a generalized version of a Lambertian diffuse shading model that tries to account for the roughness of the surface. They applied the Torrance—Sparrow model for rough surfaces with isotropic roughness and provided parameters to account for the various surface structures found in the Torrance—Sparrow model. By comparing their model with actual data, they simplified their model to the terms that had the most significant impact. The Oren—Nayar diffuse shading model looks like this.

$$i_d = \frac{\rho}{\pi} E_0 \cos(\theta_i) (A + B \max[0, \cos(\phi_r - \phi_i)] \sin(\alpha) \tan(\beta))$$

Where

$$A = 1 - 0.5 \frac{\sigma^2}{\sigma^2 + 0.33}$$

$$B = 0.45 \frac{\sigma^2}{\sigma^2 + 0.09}$$

Now this may look daunting, but it can be simplified to something we can appreciate if we replace the original notation with the notation we've already been using. ρ/π is a surface reflectivity property, which we can replace with our surface diffuse color. E_0 is a light input energy term, which we can replace with our light diffuse color. And the θ_i term is just our familiar angle between the vertex normal and the light direction. Making these exchanges gives us

$$i_d = (m_d \otimes s_d)(\hat{n} \cdot \hat{l})(A + B \max[0, \cos(\phi_r - \phi_i)] \sin(\alpha) \tan(\beta))$$

(Oren-Nayar)

Which looks a lot more like the equations we've used, there are still some parameters to explain.

σ is the surface roughness parameter. It's the standard deviation in radians of the angle of distribution of the microfacets in the surface roughness model. The larger the value, the rougher the surface.

θ_r is the angle between the vertex normal and the view direction.

$\varphi_r - \varphi_i$ is the circular angle (about the vertex normal) between the light vector and the view vector.

α is $\max(\theta_i, \theta_r)$.

β is $\min(\theta_i, \theta_r)$.

Note that if the roughness value is zero, the model is the same as the Lambertian diffuse model. Oren and Nayar also note that we can replace the value 0.33 in coefficient A with 0.57 to better account for surface inter-reflection.



Lecture No.31 Mathematics of Lighting and Shading Part V

Physically Based Illumination

In order to get a more realistic representation of lighting, we need to move away from the simplistic models that are found hard coded in most graphics pipelines and move to something that is based more in a physical representation of light as a wave with properties of its own that can interact with its environment. To do this, we'll need to understand how light passes through a medium and how hitting the boundary layer at the intersection of two media can affect light's properties. In Figure 1, there's an incident light hitting a surface. At the boundary of the two media (in this case, air and glass), there are two resulting rays of light. The reflected ray is the one that we've already discussed to some extent, and the other ray is the refracted or transmitted ray.

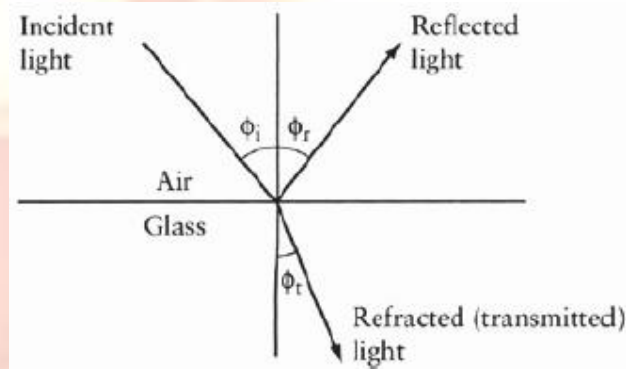


Figure 1: Light being reflected and refracted through a boundary.

In addition to examining the interaction of light with the surface boundary, we need a better description of real surface geometries. Until now, we've been treating our surfaces as perfectly smooth and uniform. Unfortunately, this prevents us from getting some interesting effects. We'll go over trying to model a real surface later, but first let's look at the physics of light interacting at a material boundary.

Reflection

Reflection of a light wave is the change in direction of the light ray when it bounces off the boundary between two media. The reflected light wave turns out to be a simple case since light is reflected at the same angle as the incident wave (when the surface is smooth and uniform, as we'll assume for now). Thus for a light wave reflecting off a perfectly smooth surface

$$\phi_{\text{incident}} = \phi_{\text{reflected}}$$

Until now, we've treated all of our specular lighting calculations as essentially reflection off a perfect surface, a surface that doesn't interact with the light in any manner other than reflecting light in proportion to the color of the surface itself. Using a lighting model based upon the Blinn—Phong model means that we'll always get a uniform specular highlight based upon the color of the reflecting light and material, which means that all reflections based on this model, will be reminiscent of plastic. In order to get a more

interesting and realistic lighting model, we need to add in some nonlinear elements to our calculations. First, let's examine what occurs when light is reflected off a surface. For a perfect reflecting surface, the angle of the incoming light (the angle of incidence) is equal to that of the reflected light. Phong's equation just blurs out the highlight a bit in a symmetrical fashion. Until we start dealing with non uniform smooth surfaces in a manner a bit more realistic than Phong's.

Refraction

Refraction happens when a light wave goes from one medium into another. Because of the difference in the speed of light of the media, light bends when it crosses the boundary. **Snell's law** gives the change in angles.

$$n_a \sin(\phi_a) = n_b \sin(\phi_b)$$

Where the n 's are the material's index of refraction,

Snell's law states that **when light refracts through a surface, the refracted angle is shifted by a function of the ratio of the two material's indices of refraction**. The index of refraction of vacuum is 1, and all other material's indices of refraction are greater than 1. What this means is that in order to realistically model refraction, we need to know the indices of refraction of the two materials that the light is traveling through. Let's look at an example (Figure 2) to see what this really means. Let's take a simple case of a ray of light traveling through the air ($n_{\text{air}} = 1$) and intersecting a glass surface ($n_{\text{glass}} = 1.5$). If the light ray hits the glass surface at 45° , at what angle does the refracted ray leave the interface?

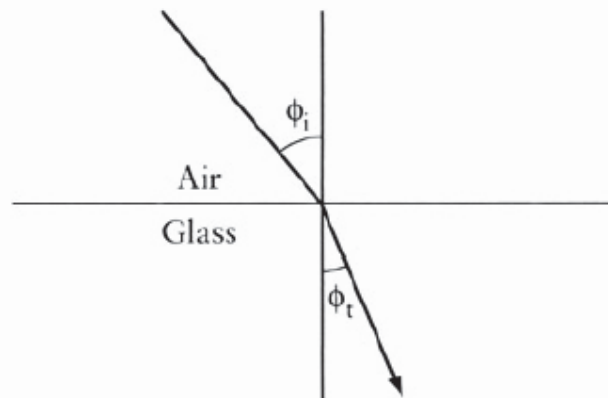


Figure 2: The refracted ray's angle is less than the incoming ray's.

The angle of incidence is the angle between the incoming vector and the surface. Rearranging Snell's law, we can solve for the refracted angle.

$$\left(\frac{n_{\text{air}}}{n_{\text{glass}}}\right) \sin(\phi_{\text{air}}) = \sin(\phi_{\text{glass}})$$

Inserting the values give us

$$\left(\frac{1.0}{1.5}\right)\sin(45^\circ) = \sin(\phi_{\text{glass}})$$

$$\phi_{\text{glass}} = 28.1^\circ$$

This is a fairly significant change in the angle! If we change things around so that we are following a light ray emerging from water into the air, we can run into another phenomenon. Since the index of refraction is just a measure of the change in speed that light travels in a material, we can observe from Snell's law (and the fact that the index of refraction in a vacuum is 1) that light bends toward the normal when it slows down (i.e., when the material it's intersecting with has a higher index of refraction). Consequently, when we intersect a medium that has a lower index of refraction (e.g., going from glass to air), then the angle will increase. Ah, we must be thinking, we're approaching a singularity here since we can then easily generate numbers that we can't take the inverse sine of! If we use Snell's law for light going from water to air, and plug in 90° for the refracted angle, we get 41.8° for the incident angle. This is called the **critical angle** at which we observe the phenomenon of *total internal reflection*. At any angle greater than this, light will not pass through a boundary but will be reflected internally. One place that we get interesting visual properties is in the diamond—air interface. The refractive index of a diamond is fairly high, 2.24, which means that it's got a very low critical angle, just 24.4° . This means that a good portion of the light entering a diamond will bounce around the inside of the diamond hitting a number of air—diamond boundaries, and as long as the angle is 24.4° or greater, it will keep reflecting internally. This is why diamonds are cut to be relatively flatish on the top but with many faceted sides, so that light entering in one spot will bounce around and exit at another, giving rise to the sparkle normally associated with diamonds.

Another place where a small change in the indices of refraction occurs is on a road heated by the sun when viewed from far away (hence a glancing incident angle). The hot air at the road's surface has a slightly smaller index of refraction than the denser, cooler air above it. This is why we get the effect of a road looking as though it were covered with water and reflecting the image above it—the light waves are actually reflected off the warm air—cold air interface.

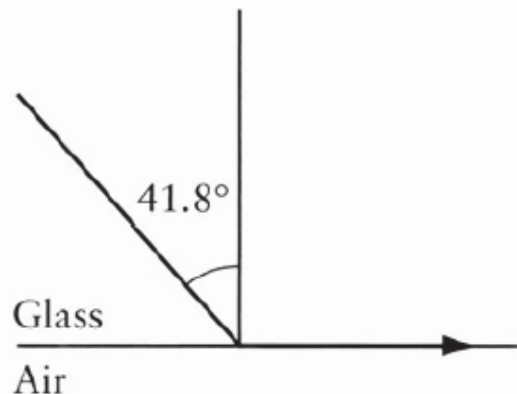


Figure 3: The critical angle.

What makes this really challenging to model is that the index of refraction for most materials is a function of the wavelength of the light. This means that not only is there a

shift in the angle of refraction, but that the shift is different for differing wavelengths of light. Figure 4 and 5 show the index of refraction for fused quartz and sapphire plotted against the wavelength. We can see the general trend that shorter wavelength light (bluish) tends to bend more than the longer (reddish) wavelengths.

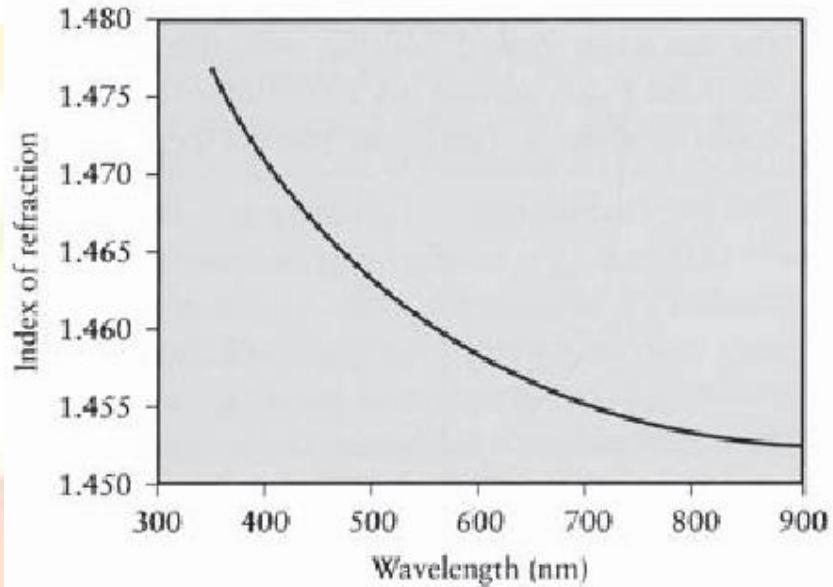


Figure 4: Index of refraction as a function of wavelength for quartz.

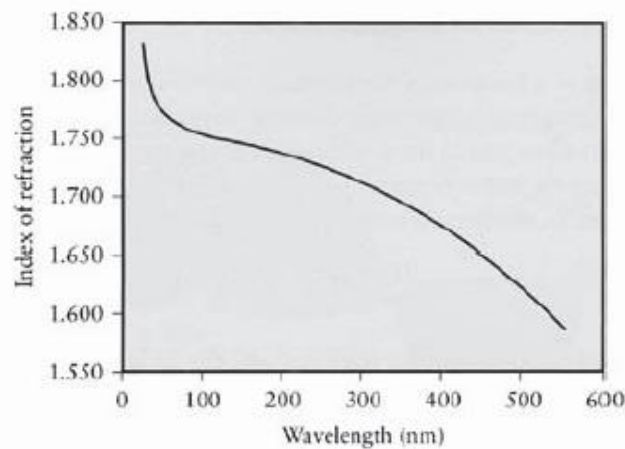


Figure 5: Index of refraction as a function of wavelength for sapphire.

This is the phenomenon that's responsible for the spectrum that can be seen when white light is passed through a prism (Figure 6). It's refraction that will break apart a light source into its component colors, not reflection.

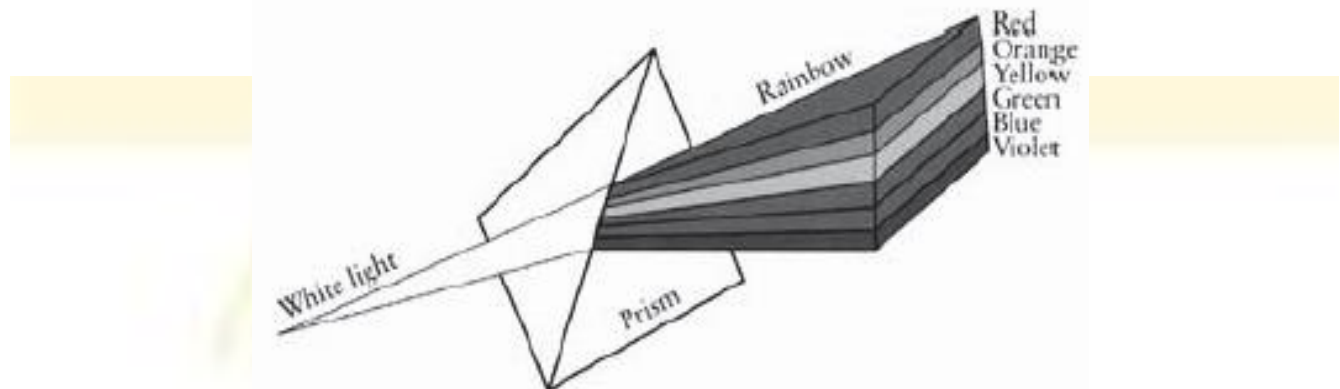


Figure 6: The wavelength dependence of the index of refraction in action.

This is one area where our simplistic model of light breaks down since we're not computing an entire spectrum of light waves, but we're limited to three primary colors. For reference, the **rgb** values can be assigned to a range of wavelengths as follows:

$$\begin{aligned}\text{blue} &= 492 - 455\text{nm} \\ \text{green} &= 577 - 492\text{nm} \\ \text{red} &= 780 - 622\text{nm}\end{aligned}$$

There's a lot more to color science than just determining wavelengths, but that's beyond our scope.

While the spectrum spreading effect of refraction is interesting in itself, the **rgb** nature of computer color representation precludes performing this spreading directly—we can't break up a color value into multiple color values. However, with some work, we can compute the shade of the color for a particular angle of refraction and then use that as the material color to influence the refracted color.

Temperature Correction for Refractive Index

Refractive index is a function of temperature, mostly due to density changes in materials with changes in temperature. A simple correction can be applied in most circumstances to allow us to use a value given at one temperature at another. For example, suppose the index of refraction value we have is given at 25°C: η_{25} . To convert the index to another temperature, η_t , we can use the following equation:

$$\eta^t = \eta^{25} + (25.0 - t)(0.00045)$$

Where the actual temperature we want is t , and the 25 is the temperature (both in °C) of the actual index we have, η_{25} .

Lecture No.32 Introduction to OpenGL

As a software interface for graphics hardware, OpenGL renders multidimensional objects into a frame buffer. OpenGL is industry-standard graphics software with which programmers can create high-quality still and animated three-dimensional color images.

Where Applicable:

OpenGL is built for compatibility across hardware and operating systems. This architecture makes it easy to port OpenGL programs from one system to another. While each operating system has unique requirements, the OpenGL code in many programs can be used as is.

Developer Audience:

Designed for use by C/C++ programmers

Run-time Requirements:

OpenGL can run on Linux and all versions of 32 bit Microsoft Windows.

Most Widely Adopted Graphics Standard

OpenGL is the premier environment for developing portable, interactive 2D and 3D graphics applications. Since its introduction in 1992, OpenGL has become the industry's most widely used and supported 2D and 3D graphics application programming interface (API), bringing thousands of applications to a wide variety of computer platforms. OpenGL fosters innovation and speeds application development by incorporating a broad set of rendering, texture mapping, special effects, and other powerful visualization functions. Developers can leverage the power of OpenGL across all popular desktop and workstation platforms, ensuring wide application deployment.

High Visual Quality and Performance

Any visual computing application requiring maximum performance-from 3D animation to CAD to visual simulation-can exploit high-quality, high-performance OpenGL capabilities. These capabilities allow developers in diverse markets such as broadcasting, CAD/CAM/CAE, entertainment, medical imaging, and virtual reality to produce and display incredibly compelling 2D and 3D graphics.

Developer-Driven Advantages

- **Industry standard**

An independent consortium, the OpenGL Architecture Review Board, guides the OpenGL specification. With broad industry support, OpenGL is the only truly open, vendor-neutral, multiplatform graphics standard.

- **Stable**

OpenGL implementations have been available for more than seven years on a wide variety of platforms. Additions to the specification are well controlled, and proposed updates are announced in time for developers to adopt changes.

Backward compatibility requirements ensure that existing applications do not become obsolete.

- **Reliable and portable**

All OpenGL applications produce consistent visual display results on any OpenGL API-compliant hardware, regardless of operating system or windowing system.

- **Evolving**

Because of its thorough and forward-looking design, OpenGL allows new hardware innovations to be accessible through the API via the OpenGL extension mechanism. In this way, innovations appear in the API in a timely fashion, letting application developers and hardware vendors incorporate new features into their normal product release cycles.

- **Scalable**

OpenGL API-based applications can run on systems ranging from consumer electronics to PCs, workstations, and supercomputers. As a result, applications can scale to any class of machine that the developer chooses to target.

- **Easy to use**

OpenGL is well structured with an intuitive design and logical commands. Efficient OpenGL routines typically result in applications with fewer lines of code than those that make up programs generated using other graphics libraries or packages. In addition, OpenGL drivers encapsulate information about the underlying hardware, freeing the application developer from having to design for specific hardware features.

- **Well-documented**

Numerous books have been published about OpenGL, and a great deal of sample code is readily available, making information about OpenGL inexpensive and easy to obtain.

Simplifies Software Development, Speeds Time-to-Market

OpenGL routines simplify the development of graphics software—from rendering a simple geometric point, line, or filled polygon to the creation of the most complex lighted and texture-mapped NURBS curved surface. OpenGL gives software developers access to geometric and image primitives, display lists, modeling transformations, lighting and texturing, anti-aliasing, blending, and many other features.

Every conforming OpenGL implementation includes the full complement of OpenGL functions. The well-specified OpenGL standard has language bindings for C, C++, Fortran, Ada, and Java. All licensed OpenGL implementations come from a single specification and language binding document and are required to pass a set of conformance tests. Applications utilizing OpenGL functions are easily portable across a wide array of platforms for maximized programmer productivity and shorter time-to-market.

All elements of the OpenGL state—even the contents of the texture memory and the frame buffer—can be obtained by an OpenGL application. OpenGL also supports visualization applications with 2D images treated as types of primitives that can be manipulated just like 3D geometric objects. As shown in the OpenGL visualization programming pipeline diagram above, images and vertices defining geometric primitives are passed through the OpenGL pipeline to the frame buffer.

Available Everywhere

Supported on all UNIX® workstations, and shipped standard with every Windows 95/98/2000/NT and MacOS PC, no other graphics API operates on a wider range of hardware platforms and software environments. OpenGL runs on every major operating system including Mac OS, OS/2, UNIX, Windows 95/98, Windows 2000, Windows NT, Linux, OPENStep, and BeOS; it also works with every major windowing system, including Win32, MacOS, Presentation Manager, and X-Window System. OpenGL is callable from Ada, C, C++, Fortran, Python, Perl and Java and offers complete independence from network protocols and topologies.

Architected for Flexibility and Differentiation!

Although the OpenGL specification defines a particular graphics processing pipeline, platform vendors have the freedom to tailor a particular OpenGL implementation to meet unique system cost and performance objectives. Individual calls can be executed on dedicated hardware, run as software routines on the standard system CPU, or implemented as a combination of both dedicated hardware and software routines. This implementation flexibility means that OpenGL hardware acceleration can range from simple rendering to full geometry and is widely available on everything from low-cost PCs to high-end workstations and supercomputers. Application developers are assured consistent display results regardless of the platform implementation of the OpenGL environment.

Using the OpenGL extension mechanism, hardware developers can differentiate their products by developing extensions that allow software developers to access additional performance and technological innovations.

Main purpose of OpenGL

As a software interface for graphics hardware, the main purpose of OpenGL is to render two- and three-dimensional objects into a frame buffer. These objects are described as sequences of vertices (that define geometric objects) or pixels (that define images). OpenGL performs several processes on this data to convert it to pixels to form the final desired image in the frame buffer.

The following topics present a global view of how OpenGL works:

- **Primitives and Commands** discusses points, line segments, and polygons as the basic units of drawing; and the processing of commands.
- **OpenGL Graphic Control** describes which graphic operations OpenGL controls and which it does not control.
- **Execution Model** discusses the client/server model for interpreting OpenGL commands.

- **Basic OpenGL Operation** gives a high-level description of how OpenGL processes data to produce a corresponding image in the frame buffer.

Primitives and Commands

OpenGL draws *primitive* points, line segments, or polygons subject to several selectable modes. You can control modes independently of one another. That is, setting one mode doesn't affect whether other modes are set (although many modes may interact to determine what eventually ends up in the frame buffer). To specify primitives, set modes, and perform other OpenGL operations, you issue commands in the form of function calls.

Primitives are defined by a group of one or more *vertices*. A vertex defines a point, an endpoint of a line, or a corner of a polygon where two edges meet. Data (consisting of vertex coordinates, colors, normals, texture coordinates, and edge flags) is associated with a vertex, and each vertex and its associated data are processed independently, in order, and in the same way. The only exceptions to this rule are cases in which the group of vertices must be clipped so that a particular primitive fits within a specified region. In this case, vertex data may be modified and new vertices created. The type of clipping depends on which primitive the group of vertices represents.

Commands are always processed in the order in which they are received, although there may be an indeterminate delay before a command takes effect. This means that each primitive is drawn completely before any subsequent command takes effect. It also means that state-querying commands return data that is consistent with complete execution of all previously issued OpenGL commands.

OpenGL Graphic Control

OpenGL provides you with fairly direct control over the fundamental operations of two- and three-dimensional graphics. This includes specification of such parameters as transformation matrices, lighting equation coefficients, antialiasing methods, and pixel-update operators. However, it doesn't provide you with a means for describing or modeling complex geometric objects. Thus, the OpenGL commands you issue specify how a certain result should be produced (what procedure should be followed) rather than what exactly that result should look like. That is, OpenGL is fundamentally procedural rather than descriptive. To fully understand how to use OpenGL, it helps to know the order in which it carries out its operations.

Execution Model

The model for interpretation of OpenGL commands is client/server. Application code (the client) issues commands, which are interpreted and processed by OpenGL (the server). The server may or may not operate on the same computer as the client. In this sense, OpenGL is network-transparent. A server can maintain several OpenGL contexts, each of which is an encapsulated OpenGL state. A client can connect to any one of these contexts. The required network protocol can be implemented by augmenting an already existing protocol (such as that of the X Window System) or by using an independent protocol. No OpenGL commands are provided for obtaining user input.

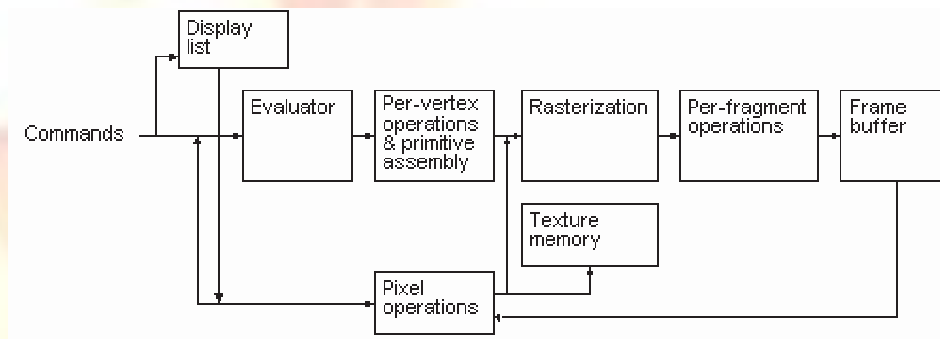
The window system that allocates frame buffer resources ultimately controls the effects of OpenGL commands on the frame buffer. The window system:

- Determines which portions of the frame buffer OpenGL may access at any given time.
- Communicates to OpenGL how those portions are structured.

Therefore, there are no OpenGL commands to configure the frame buffer or initialize OpenGL. Frame buffer configuration is done outside of OpenGL in conjunction with the window system; OpenGL initialization takes place when the window system allocates a window for OpenGL rendering.

Basic OpenGL Operation

The following diagram illustrates how OpenGL processes data. As shown, commands enter from the left and proceed through a processing pipeline. Some commands specify geometric objects to be drawn, and others control how the objects are handled during various processing stages.



The processing stages in basic OpenGL operation are as follows:

- **Display list** Rather than having all commands proceed immediately through the pipeline, you can choose to accumulate some of them in a display list for processing later.
- **Evaluator** The evaluator stage of processing provides an efficient way to approximate curve and surface geometry by evaluating polynomial commands of input values.
- **Per-vertex operations and primitive assembly** OpenGL processes geometric primitives—points, line segments, and polygons—all of which are described by vertices. Vertices are transformed and lit, and primitives are clipped to the view port in preparation for rasterization.
- **Rasterization** The rasterization stage produces a series of frame-buffer addresses and associated values using a two-dimensional description of a point, line segment, or polygon. Each fragment so produced is fed into the last stage, per-fragment operations.
- **Per-fragment operations** these are the final operations performed on the data before it's stored as pixels in the frame buffer.

Per-fragment operations include conditional updates to the frame buffer based on incoming and previously stored z values (for z buffering) and blending of

incoming pixel colors with stored colors, as well as masking and other logical operations on pixel values.

Data can be input in the form of pixels rather than vertices. Data in the form of pixels, such as might describe an image for use in texture mapping, skips the first stage of processing described above and instead is processed as pixels, in the pixel operations stage. Following pixel operations, the pixel data is either:

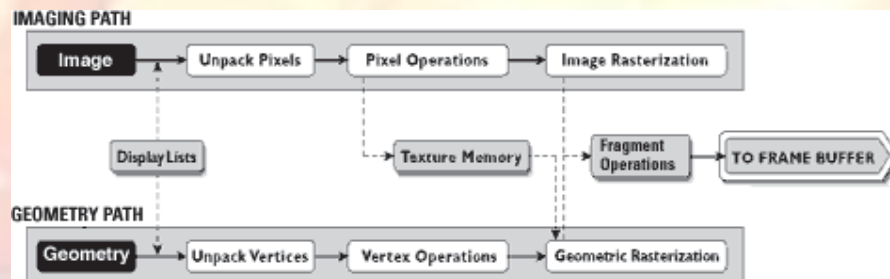
- Stored as texture memory, for use in the rasterization stage.
- Rasterized, with the resulting fragments merged into the frame buffer just as if they were generated from geometric data.

OpenGL Processing Pipeline

Many OpenGL functions are used specifically for drawing objects such as points, lines, polygons, and bitmaps. Some functions control the way that some of this drawing occurs (such as those that enable antialiasing or texturing). Other functions are specifically concerned with frame buffer manipulation. The topics in this section describe how all of the OpenGL functions work together to create the OpenGL processing pipeline. This section also takes a closer look at the stages in which data is actually processed, and ties these stages to OpenGL functions.

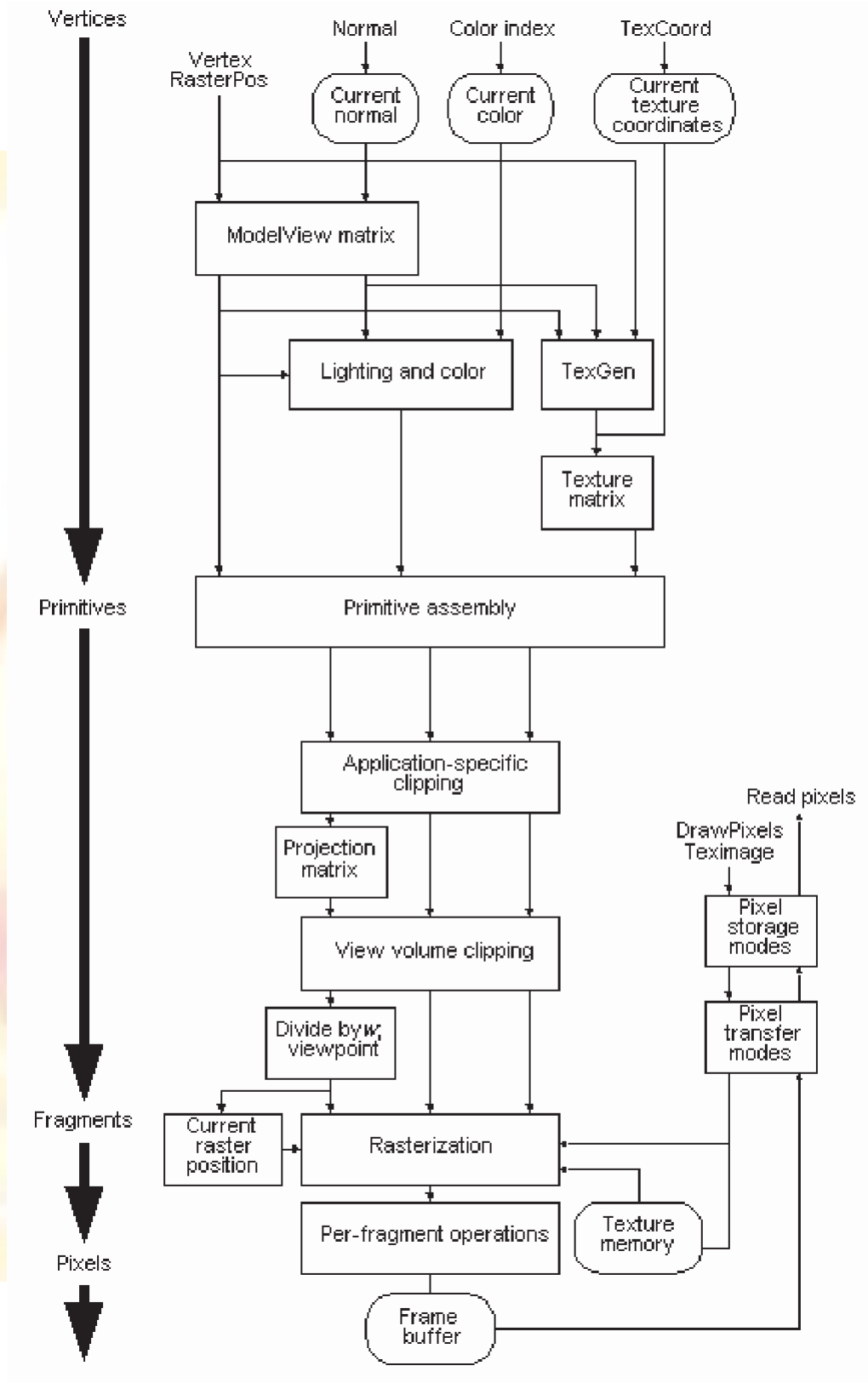
The following diagram details the OpenGL processing pipeline. For most of the pipeline, you can see three vertical arrows between the major stages. These arrows represent vertices and the two primary types of data that can be associated with vertices: color values and texture coordinates. Also note that vertices are assembled into primitives, then into fragments, and finally into pixels in the framebuffer.

The OpenGL Visualization Programming Pipeline



OpenGL operates on image data as well as geometric primitives.

A detailed view on the next page.



Lecture No.33 OpenGL Programming - I

For writing a program, using OpenGL, for any of the operating systems we can use OpenGL Utility Library named, “glut”. “glut” is used to initialize OpenGL on any platform e.g. Microsoft Windows or Linux etc because it is platform independent. “glut” can create window, get keyboard input and run an event handler or message loop in our graphics application.

All those functions that start with the prefix “gl” are the core OpenGL functions and those which start with “glu” or “glut” are the OpenGL utility library functions.

Let’s write a program that uses “glut” and then uses OpenGL function to create graphics.

```
#include <GL/glut.h>
int main()
{
    glutCreateWindow( "first graphics window" );
}
```

glutCreateWindow
glutCreateWindow creates a top-level window.

Usage

```
int glutCreateWindow(char *name);
name:
```

ASCII character string for use as window name.

Description: glutCreateWindow creates a top-level window. The name will be provided to the window system as the window's name. The intent is that the window system will label the window with the name.

Implicitly, the *current window* is set to the newly created window. Each created window has a unique associated OpenGL context. State changes to a window's associated OpenGL context can be done immediately after the window is created.

The *display state* of a window is initially for the window to be shown. But the window's *display state* is not actually acted upon until glutMainLoop is entered. This means until glutMainLoop is called, rendering to a created window is ineffective because the window cannot yet be displayed.

The value returned is a unique small integer identifier for the window. The range of allocated identifiers starts at one. This window identifier can be used when calling glutSetWindow.

X Implementation Notes

The proper X Inter-Client Communication Conventions Manual (ICCCM) top-level properties are established. The WM_COMMAND property that lists the command line used to invoke the GLUT program is only established for the first window created.

This is the simple program that we have written so far. Now we will use more features of “glut” library.

```
#include <GL/glut.h>
int main()
{
    //Set up the OpenGL rendering context.
    glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );

    //Create a window and set its width and height.
    glutCreateWindow( "Deform" );
    glutReshapeWindow( 640, 480 );

    //The keyboard function gets called whenever we hit a key.
    glutKeyboardFunc( keyboard );

    //The display function gets called whenever the window
    //requires an update or when we explicitly call
    glutDisplayFunc( display );

    //The reshape function gets called whenever the window changes
    //shape.
    glutReshapeFunc( reshape );

    //The idle function gets called when there are no window-
    //events to process.
    glutIdleFunc( idle );

    //Get the ball rolling!
    glutMainLoop();
}
```

In the above program we have used “glut” functions. Let us discuss them in detail.

GlutInitDisplayMode

glutInitDisplayMode sets the *initial display mode*.

Usage

```
void glutInitDisplayMode(unsigned int mode);
mode
```

Display mode, normally the bitwise *OR*-ing of GLUT display mode bit masks. See values below:

GLUT_RGBA

Bit mask to select an RGBA mode window. This is the default if neither GLUT_RGBA nor GLUT_INDEX are specified.

GLUT_RGB

An alias for GLUT_RGBA.

GLUT_INDEX

Bit mask to select a color index mode window. This overrides GLUT_RGBA if it is also specified.

GLUT_SINGLE

Bit mask to select a single buffered window. This is the default if neither GLUT_DOUBLE or GLUT_SINGLE are specified.

GLUT_DOUBLE

Bit mask to select a double buffered window. This overrides GLUT_SINGLE if it is also specified.

GLUT_ACCUM

Bit mask to select a window with an accumulation buffer.

GLUT_ALPHA

Bit mask to select a window with an alpha component to the color buffer(s).

GLUT_DEPTH

Bit mask to select a window with a depth buffer.

GLUT_STENCIL

Bit mask to select a window with a stencil buffer.

GLUT_MULTISAMPLE

Bit mask to select a window with multisampling support. If multisampling is not available, a non-multisampling window will automatically be chosen. Note: both the OpenGL client-side and server-side implementations must support the GLX_SAMPLE_SGIS extension for multisampling to be available.

GLUT_STEREO

Bit mask to select a stereo window.

GLUT_LUMINANCE

Bit mask to select a window with a "luminance" color model. This model provides the functionality of OpenGL's RGBA color model, but the green and blue components are not maintained in the frame buffer. Instead each pixel's red component is converted to an index between zero and `glutGet(GLUT_WINDOW_COLORMAP_SIZE)-1` and looked up in a per-window color map to determine the color of pixels within the window. The initial colormap of GLUT_LUMINANCE windows is initialized to be a linear gray ramp, but can be modified with GLUT's colormap routines.

Description

The *initial display mode* is used when creating top-level windows, subwindows, and overlays to determine the OpenGL display mode for the to-be-created window or overlay.

Note that GLUT_RGBA selects the RGBA color model, but it does not request any bits of alpha (sometimes called an *alpha buffer* or *destination alpha*) be allocated. To request alpha, specify GLUT_ALPHA. The same applies to GLUT_LUMINANCE.

GLUT_LUMINANCE Implementation Notes

GLUT_LUMINANCE is not supported on most OpenGL platforms.

glutReshapeWindow

`glutReshapeWindow` requests a change to the size of the *current window*.

Usage

```
void glutReshapeWindow(int width, int height);  
width
```

New width of window in pixels.

height

New height of window in pixels.

Description

glutReshapeWindow requests a change in the size of the *current window*. The width and height parameters are size extents in pixels. The width and height must be positive values.

The requests by glutReshapeWindow are not processed immediately. The request is executed after returning to the main event loop. This allows multiple glutReshapeWindow, glutPositionWindow, and glutFullScreen requests to the same window to be coalesced.

In the case of top-level windows, a glutReshapeWindow call is considered only to be a request for sizing the window. The window system is free to apply its own policies to top-level window sizing. The intent is that top-level windows should be reshaped according to glutReshapeWindow's parameters. Whether a reshape actually takes effect and, if so, the reshaped dimensions are reported to the program by a reshape callback.

glutReshapeWindow disables the full screen status of a window if previously enabled.

glutKeyboardFunc

glutKeyboardFunc sets the keyboard callback for the *current window*.

Usage

```
void glutKeyboardFunc(void (*func)(unsigned char key,  
int x, int y));
```

func

The new keyboard callback function.

Description

glutKeyboardFunc sets the keyboard callback for the *current window*. When a user types into the window, each key press generating an ASCII character will generate a keyboard callback. The key callback parameter is the generated ASCII character. The state of modifier keys such as Shift cannot be determined directly; their only effect will be on the returned ASCII data. The x and y callback parameters indicate the mouse location in window relative coordinates when the key was pressed. When a new window is created, no keyboard callback is initially registered, and ASCII key strokes in the window are ignored. Passing NULL to glutKeyboardFunc disables the generation of keyboard callbacks.

During a keyboard callback, glutGetModifiers may be called to determine the state of modifier keys when the keystroke generating the callback occurred.

We can also see glutSpecialFunc for a means to detect non-ASCII key strokes.

glutDisplayFunc

glutDisplayFunc sets the display callback for the *current window*.

Usage

```
void glutDisplayFunc(void (*func)(void));
```

func

The new display callback function.

Description

glutDisplayFunc sets the display callback for the *current window*. When GLUT determines that the normal plane for the window needs to be redisplayed, the display callback for the window is called. Before the callback, the *current window* is set to the window needing to be redisplayed and (if no overlay display callback is registered) the *layer in use* is set to the normal plane. The display callback is called with no parameters. The entire normal plane region should be redisplayed in response to the callback (this includes ancillary buffers if your program depends on their state).

GLUT determines when the display callback should be triggered based on the window's redisplay state. The redisplay state for a window can be either set explicitly by calling glutPostRedisplay or implicitly as the result of window damage reported by the window system. Multiple posted redisplays for a window are coalesced by GLUT to minimize the number of display callbacks called.

When an overlay is established for a window, but there is no overlay display callback registered, the display callback is used for redisplaying *both* the overlay and normal plane (that is, it will be called if either the redisplay state or overlay redisplay state is set). In this case, the *layer in use* is *not* implicitly changed on entry to the display callback.

See glutOverlayDisplayFunc to understand how distinct callbacks for the overlay and normal plane of a window may be established.

When a window is created, no display callback exists for the window. It is the responsibility of the programmer to install a display callback for the window before the window is shown. A display callback *must* be registered for any window that is shown. If a window becomes displayed without a display callback being registered, a fatal error occurs. Passing NULL to glutDisplayFunc is illegal as of GLUT 3.0; there is no way to "deregister" a display callback (though another callback routine can always be registered).

Upon return from the display callback, the *normal damaged* state of the window (returned by calling glutLayerGet(GLUT_NORMAL_DAMAGED)) is cleared. If there is no overlay display callback registered the *overlay damaged* state of the window (returned by calling glutLayerGet(GLUT_OVERLAY_DAMAGED)) is also cleared.

glutReshapeFunc

glutReshapeFunc sets the reshape callback for the *current window*.

Usage

```
void glutReshapeFunc(void (*func)(int width, int height));
```

func

The new reshape callback function.

Description

glutReshapeFunc sets the reshape callback for the *current window*. The reshape callback is triggered when a window is reshaped. A reshape callback is also triggered immediately before a window's first display callback after a window is created or whenever an overlay for the window is established. The width and height parameters of the callback specify the new window size in pixels. Before the callback, the *current window* is set to the window that has been reshaped.

If a reshape callback is not registered for a window or NULL is passed to glutReshapeFunc (to deregister a previously registered callback), the default reshape

callback is used. This default callback will simply call `glViewport(0,0,width,height)` on the normal plane (and on the overlay if one exists).

If an overlay is established for the window, a single reshape callback is generated. It is the callback's responsibility to update both the normal plane and overlay for the window (changing the *layer in use* as necessary).

When a top-level window is reshaped, subwindows are not reshaped. It is up to the GLUT program to manage the size and positions of subwindows within a top-level window. Still, reshape callbacks will be triggered for subwindows when their size is changed using `glutReshapeWindow`.

`glutIdleFunc`

`glutIdleFunc` sets the global idle callback.

Usage

```
void glutIdleFunc(void (*func)(void));
```

Description

`glutIdleFunc` sets the global idle callback to be 'func' so a GLUT program can perform background processing tasks or continuous animation when window system events are not being received. If enabled, the idle callback is continuously called when events are not being received. The callback routine has no parameters. The *current window* and *current menu* will not be changed before the idle callback. Programs with multiple windows and/or menus should explicitly set the *current window* and/or *current menu* and not rely on its current setting.

The amount of computation and rendering done in an idle callback should be minimized to avoid affecting the program's interactive response. In general, not more than a single frame of rendering should be done in an idle callback.

Passing NULL to `glutIdleFunc` disables the generation of the idle callback.

`glutMainLoop`

`glutMainLoop` enters the GLUT event processing loop.

Usage

```
void glutMainLoop(void);
```

Description

`glutMainLoop` enters the GLUT event processing loop. This routine should be called at most once in a GLUT program. Once called, this routine will never return. It will call as necessary any callbacks that have been registered.

`glutSwapBuffers`

`glutSwapBuffers` swaps the buffers of the *current window* if double buffered.

Usage

```
void glutSwapBuffers(void);
```

Description

Performs a buffer swap on the *layer in use* for the *current window*. Specifically, `glutSwapBuffers` promotes the contents of the back buffer of the *layer in use* of the *current window* to become the contents of the front buffer. The contents of the back buffer then become undefined. The update typically takes place during the vertical retrace of the monitor, rather than immediately after `glutSwapBuffers` is called.

An implicit `glFlush` is done by `glutSwapBuffers` before it returns. Subsequent OpenGL commands can be issued immediately after calling `glutSwapBuffers`, but are not executed until the buffer exchange is completed.

If the *layer in use* is not double buffered, `glutSwapBuffers` has no effect.

Lecture No.34 OpenGL Programming - II

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

static void reshape( int w, int h )
{
    glViewport( 0, 0, w, h );
}

static void keyboard( unsigned char key, int x, int y )
{
    switch( key )
    {
        case 27: //Escape
            exit(0);
            break;
    }
}

static void display()
{
}

static void idle()
{
    glClearColor(0.0f,0.0f,0.0f,0.0f);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    ////////////////////////////////////////
    // glRotatef(0.5f,0.0f,1.0f,0.0);

    glBegin(GL_TRIANGLES);

    glVertex3f(0.5f,0.2f,0.0f);
    glVertex3f(0.5f,0.0f,0.0f);
    glVertex3f(0.0f,0.0f,0.0f);

    glEnd();

    glutSwapBuffers();
}

static void initGL()
{
    float ratio= (float)640 /480 ;

    glMatrixMode( GL_PROJECTION );
    glLoadIdentity();
```

```
        gluPerspective( 45.0,(float) 640 /480 , 1.0, 100.0 );
        glMatrixMode(GL_MODELVIEW);

        glLoadIdentity();
        glTranslated(0.0, 0.0, -5.0);

    }
int main( int argc, char* argv[] )
{

    //Initialize the GLUT library.
    glutInit( &argc, argv );

    //Set up the OpenGL rendering context.
    glutInitDisplayMode( GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH );

    //Create a window and set its width and height.
    glutCreateWindow( "Deform" );
    glutReshapeWindow( 640, 480 );

    //Initialize our data structures.
    initGL();

    //The keyboard function gets called whenever we hit a key.
    glutKeyboardFunc( keyboard );

    //The display function gets called whenever the window
    //requires an update or when we explicitly call
    //glutPostRedisplay()
    glutDisplayFunc( display );

    //The reshape function gets called whenever the window changes
    //shape.
    glutReshapeFunc( reshape );

    //The idle function gets called when there are no window
    //events to process.
    glutIdleFunc( idle );

    //Get the ball rolling!
    glutMainLoop();

    return 0;
}

//eof
```

In the above program we have first set perspective projection matrix and then rendered a triangle in idle function.

`glMatrixMode`

The **glMatrixMode** function specifies which matrix is the current matrix.

```
void glMatrixMode(
    GLenum mode
);
```

Parameters

mode

The matrix stack that is the target for subsequent matrix operations. The *mode* parameter can assume one of three values:

Value	Meaning
GL_MODELVIEW	Applies subsequent matrix operations to the modelview matrix stack.
GL_PROJECTION	Applies subsequent matrix operations to the projection matrix stack.
GL_TEXTURE	Applies subsequent matrix operations to the texture matrix stack.

Remarks

The **glMatrixMode** function sets the current matrix mode.

The following function retrieves information related to **glMatrixMode**:

Error Codes

The following are the error codes generated and their conditions.

Error code	Condition
GL_INVALID_ENUM	<i>mode</i> was not an accepted value.
GL_INVALID_OPERATION	glMatrixMode was called between a call to glBegin and the corresponding call to glEnd .

`glLoadIdentity`

The **glLoadIdentity** function replaces the current matrix with the identity matrix.

```
void glLoadIdentity(
    void
);
```

Remarks

The **glLoadIdentity** function replaces the current matrix with the identity matrix. It is semantically equivalent to calling **glLoadMatrix** with the identity matrix

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

but in some cases it is more efficient.

The following functions retrieve information related to **glLoadIdentity**:

Error Codes

The following is the error code and its condition.

Error code	Condition
GL_INVALID_OPERATION	glLoadIdentity was called between a call to glBegin and the corresponding call to glEnd .

glTranslated, **glTranslatef**

The **glTranslated** and **glTranslatef** functions multiply the current matrix by a translation matrix.

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

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

Parameters *x*, *y*, *z*

The *x*, *y*, and *z* coordinates of a translation vector.

Remarks

The **glTranslate** function produces the translation specified by (*x*, *y*, *z*). The translation vector is used to compute a 4x4 translation matrix:

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

The current matrix (see **glMatrixMode**) is multiplied by this translation matrix, with the product replacing the current matrix. That is, if *M* is the current matrix and *T* is the translation matrix, then *M* is replaced with *M*•*T*.

If the matrix mode is either **GL_MODELVIEW** or **GL_PROJECTION**, all objects drawn after **glTranslate** is called are translated. Use **glPushMatrix** and **glPopMatrix** to save and restore the untranslated coordinate system.

The following functions retrieve information related to **glTranslated** and **glTranslatef**:

Error Codes

The following is the error code and its condition.

Error code	Condition
GL_INVALID_OPERATION	glTranslate was called between a call to glBegin and the corresponding call to glEnd .

`gluPerspective`

The **gluPerspective** is the function of gl utility library. This function sets up a perspective projection matrix.

```
void gluPerspective(
    GLdouble fovy,
    GLdouble aspect,
    GLdouble zNear,
    GLdouble zFar
);
```

Parameters

fovy

The field of view angle, in degrees, in the y-direction.

aspect

The aspect ratio that determines the field of view in the x-direction. The aspect ratio is the ratio of x (width) to y (height).

zNear

The distance from the viewer to the near clipping plane (always positive).

zFar

The distance from the viewer to the far clipping plane (always positive).

Remarks

The **gluPerspective** function specifies a viewing frustum into the world coordinate system. In general, the aspect ratio in **gluPerspective** should match the aspect ratio of the associated viewport. For example, *aspect* = 2.0 means the viewer's angle of view is twice as wide in x as it is in y. If the viewport is twice as wide as it is tall, it displays the image without distortion.

The matrix generated by **gluPerspective** is multiplied by the current matrix, just as if **glMultMatrix** were called with the generated matrix. To load the perspective matrix onto the current matrix stack instead, precede the call to **gluPerspective** with a call to **glLoadIdentity**.

`glRotated`, `glRotatef`

The **glRotated** and **glRotatef** functions multiply the current matrix by a rotation matrix.

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

```

GLfloat x,
GLfloat y,
GLfloat z );
    Parameters

```

angle

The angle of rotation, in degrees.

x, y, z

The *x*, *y*, and *z* coordinates of a vector, respectively.

Remarks

The **glRotate** function computes a matrix that performs a counterclockwise rotation of *angle* degrees about the vector from the origin through the point (*x*, *y*, *z*).

The current matrix (see **glMatrixMode**) is multiplied by this rotation matrix, with the product replacing the current matrix. That is, if *M* is the current matrix and *R* is the translation matrix, then *M* is replaced with *M*•*R*.

If the matrix mode is either `GL_MODELVIEW` or `GL_PROJECTION`, all objects drawn after **glRotate** is called are rotated. Use **glPushMatrix** and **glPopMatrix** to save and restore the unrotated coordinate system.

Error Codes

The following is the error code and its condition.

Error code	Condition
<code>GL_INVALID_OPERATION</code>	glRotate was called between a call to glBegin and the corresponding call to glEnd .

glClearColor

The **glClearColor** function specifies clear values for the color buffers.

```

void glClearColor(
    GLclampf red,
    GLclampf green,
    GLclampf blue,
    GLclampf alpha
);

```

Parameters

red, *green*, *blue*, *alpha*

The red, green, blue, and alpha values that **glClear** uses to clear the color buffers. The default values are all zero.

Remarks

The **glClearColor** function specifies the red, green, blue, and alpha values used by **glClear** to clear the color buffers. Values specified by **glClearColor** are clamped to the range [0,1].

Error Codes

The following is the error code generated and its condition.

Error code	Condition
------------	-----------

GL_INVALID_OPERATION

glClearColor was called between a call to **glBegin** and the corresponding call to **glEnd**.

glColor

These functions set the current color.

glColor3b ,	glColor3d ,	glColor3f ,	glColor3i ,	glColor3s ,
glColor3ub ,	glColor3ui ,	glColor3us ,	glColor4b ,	glColor4d ,
glColor4f ,	glColor4i ,	glColor4s ,	glColor4ub ,	glColor4ui ,
glColor4us ,	glColor3bv ,	glColor3dv ,	glColor3fv ,	glColor3iv ,
glColor3sv ,	glColor3ubv ,	glColor3uiv ,	glColor3usv ,	glColor3sv ,
glColor4bv ,	glColor4dv ,	glColor4fv ,	glColor4iv ,	glColor4sv ,
glColor4ubv ,	glColor4uiv ,	glColor4usv .		

void glColor3b(

GLbyte *red*,

GLbyte *green*,

GLbyte *blue*);

void glColor3f(

GLfloat *red*,

GLfloat *green*,

GLfloat *blue*);

Consult yourself for the documentation of rest of the functions of this type.

Parameters

red, green, blue

New red, green, and blue values for the current color.

alpha

A new alpha value for the current color. Included only in the four-argument **glColor4** function.

Remarks

OpenGL stores both a current single-valued color index and a current four-valued RGBA color. The **glColor** function sets a new four-valued RGBA color.

There are two major variants to **glColor**:

- The **glColor3** variants specify new red, green, and blue values explicitly, and set the current alpha value to 1.0 implicitly.
- The **glColor4** variants specify all four color components explicitly.

The **glColor3b**, **glColor4b**, **glColor3s**, **glColor4s**, **glColor3i**, and **glColor4i** functions take three or four signed byte, short, or long integers as arguments. When you append *v* to the name, the color functions can take a pointer to an array of such values.

Current color values are stored in floating-point format, with unspecified mantissa and exponent sizes. Unsigned integer color components, when specified, are linearly mapped to floating-point values such that the largest representable value maps to 1.0 (full intensity), and zero maps to 0.0 (zero intensity). Signed integer color components, when specified, are linearly mapped to floating-point values such that the most positive representable value maps to 1.0, and the most negative representable value maps to -1.0. Floating-point values are mapped directly.

Neither floating-point nor signed integer values are clamped to the range [0,1] before updating the current color. However, color components are clamped to this range before they are interpolated or written into a color buffer.

You can update the current color at any time. In particular, you can call **glColor** between a call to **glBegin** and the corresponding call to **glEnd**.

glClear

The **glClear** function clears buffers to preset values.

```
void glClear(
    GLbitfield mask
);
```

Parameters

mask

Bitwise OR operators of masks that indicate the buffers to be cleared. The four masks are as follows.

Mask	Buffer to be Cleared
GL_COLOR_BUFFER_BIT	The buffers currently enabled for color writing.
GL_DEPTH_BUFFER_BIT	The depth buffer.
GL_ACCUM_BUFFER_BIT	The accumulation buffer.
GL_STENCIL_BUFFER_BIT	The stencil buffer.

Remarks

The **glClear** function sets the bitplane area of the window to values previously selected by **glClearColor**, **glClearIndex**, **glClearDepth**, **glClearStencil**, and **glClearAccum**. You can clear multiple color buffers simultaneously by selecting more than one buffer at a time using **glDrawBuffer**.

The pixel-ownership test, the scissor test, dithering, and the buffer writemasks affect the operation of **glClear**. The scissor box bounds the cleared region. The **glClear** function ignores the alpha function, blend function, logical operation, stenciling, texture mapping, and z-buffering.

The **glClear** function takes a single argument (*mask*) that is the bitwise OR of several values indicating which buffer is to be cleared.

The value to which each buffer is cleared depends on the setting of the clear value for that buffer.

If a buffer is not present, a **glClear** call directed at that buffer has no effect.

Error Codes

The following are the error codes generated and their conditions.

Error code	Condition
GL_INVALID_VALUE	Any bit other than the four defined bits was set in <i>mask</i> .
GL_INVALID_OPERATION	glClear was called between a call to glBegin and the corresponding call to glEnd .

glBegin, glEnd

The **glBegin** and **glEnd** functions delimit the vertices of a primitive or a group of like primitives.

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

Parameters

mode

The primitive or primitives that will be created from vertices presented between **glBegin** and the subsequent **glEnd**. The following are accepted symbolic constants and their meanings:

GL_POINTS

Treats each vertex as a single point. Vertex n defines point n . N points are drawn.

GL_LINES

Treats each pair of vertices as an independent line segment. Vertices $2n - 1$ and $2n$ define line n . $N/2$ lines are drawn.

GL_LINE_STRIP

Draws a connected group of line segments from the first vertex to the last. Vertices n and $n + 1$ define line n . $N - 1$ lines are drawn.

GL_LINE_LOOP

Draws a connected group of line segments from the first vertex to the last, then back to the first. Vertices n and $n + 1$ define line n . The last line, however, is defined by vertices N and 1 . N lines are drawn.

GL_TRIANGLES

Treats each triplet of vertices as an independent triangle. Vertices $3n - 2$, $3n - 1$, and $3n$ define triangle n . $N/3$ triangles are drawn.

GL_TRIANGLE_STRIP

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. For odd n , vertices n , $n + 1$, and $n + 2$ define triangle n . For even n , vertices $n + 1$, n , and $n + 2$ define triangle n . $N - 2$ triangles are drawn.

GL_TRIANGLE_FAN

Draws a connected group of triangles. One triangle is defined for each vertex presented after the first two vertices. Vertices 1 , $n + 1$, and $n + 2$ define triangle n . $N - 2$ triangles are drawn.

GL_QUADS

Treats each group of four vertices as an independent quadrilateral. Vertices $4n - 3$, $4n - 2$, $4n - 1$, and $4n$ define quadrilateral n . $N/4$ quadrilaterals are drawn.

GL_QUAD_STRIP

Draws a connected group of quadrilaterals. One quadrilateral is defined for each pair of vertices presented after the first pair. Vertices $2n - 1$, $2n$, $2n + 1$, and $2n + 2$ define quadrilateral n . N quadrilaterals are drawn. Note that the order in which vertices are used to construct a quadrilateral from strip data is different from that used with independent data.

GL_POLYGON

Draws a single, convex polygon. Vertices 1 through N define this polygon.

Remarks

The **glBegin** and **glEnd** functions delimit the vertices that define a primitive or a group of like primitives. The **glBegin** function accepts a single argument that specifies which of ten primitives the vertices compose. Taking n as an integer count starting at one, and N as the total number of vertices specified, the interpretations are as follows:

- You can use only a subset of OpenGL functions between **glBegin** and **glEnd**. The functions you can use are:

```
glVertex
glColor
glIndex
glNormal
glMaterial
```

You can also use **glCallList** or **glCallLists** to execute display lists that include only the preceding functions. If any other OpenGL function is called between **glBegin** and **glEnd**, the error flag is set and the function is ignored.

- Regardless of the value chosen for *mode* in **glBegin**, there is no limit to the number of vertices you can define between **glBegin** and **glEnd**. Lines, triangles, quadrilaterals, and polygons that are incompletely specified are not drawn. Incomplete specification results when either too few vertices are provided to specify even a single primitive or when an incorrect multiple of vertices is specified. The incomplete primitive is ignored; the complete primitives are drawn.
- The minimum specification of vertices for each primitive is:

Minimum number of vertices	Type of primitive
1	Point
2	Line
3	Triangle
4	Quadrilateral
3	Polygon

Modes that require a certain multiple of vertices are `GL_LINES` (2), `GL_TRIANGLES` (3), `GL_QUADS` (4), and `GL_QUAD_STRIP` (2).

Error Codes

The following are the error codes generated and their conditions.

Error code	Condition
<code>GL_INVALID_ENUM</code>	<i>mode</i> was set to an unaccepted value.
<code>GL_INVALID_OPERATION</code>	A function other than <code>glVertex</code> , <code>glColor</code> , <code>glIndex</code> , <code>glNormal</code> , <code>glTexCoord</code> , <code>glEvalCoord</code> , <code>glEvalPoint</code> , <code>glMaterial</code> , <code>glEdgeFlag</code> , <code>glCallList</code> , or <code>glCallLists</code> was called between <code>glBegin</code> and the corresponding <code>glEnd</code> . The function <code>glEnd</code> was called before the corresponding <code>glBegin</code> was called, or <code>glBegin</code> was called within a <code>glBegin/glEnd</code> sequence.

glVertex

These functions specify a vertex.

```
glVertex2d,    glVertex2f,    glVertex2i,    glVertex2s,    glVertex3d,
glVertex3f,    glVertex3i,    glVertex3s,    glVertex4d,    glVertex4f,
glVertex4i,    glVertex4s,    glVertex2dv,   glVertex2fv,   glVertex2iv,
glVertex2sv,   glVertex3dv,   glVertex3fv,   glVertex3iv,   glVertex3sv,
glVertex4dv,  glVertex4fv,  glVertex4iv,  glVertex4sv
```

```
void glVertex3f(
    GLfloat x,
    GLfloat y,
    GLfloat z
);
```

Consult the documentation to yourself for the functions of the same nature.

Parameters

x, *y*, *z*, *w*

The *x*, *y*, *z*, and *w* coordinates of a vertex. Not all parameters are present in all forms of the command.

Remarks

The `glVertex` function commands are used within `glBegin/glEnd` pairs to specify point, line, and polygon vertices. The current color, normal, and texture coordinates are associated with the vertex when `glVertex` is called.

When only *x* and *y* are specified, *z* defaults to 0.0 and *w* defaults to 1.0. When *x*, *y*, and *z* are specified, *w* defaults to 1.0.

Invoking `glVertex` outside of a `glBegin/glEnd` pair results in undefined behavior.

Lecture No.35 Curves

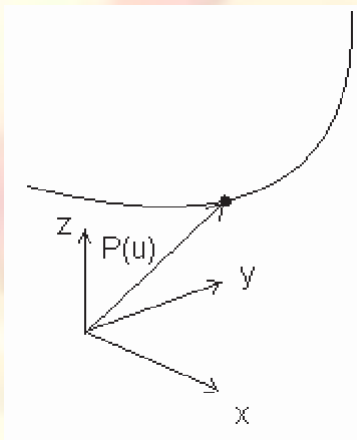
We all know what a curve is. In this lecture we will explore the mathematical definition of a curve in a form that is very useful to geometric modeling and other computer graphics applications: that definition consists of a set of parametric equations. The mathematics of parametric equations is the basis for Bezier, NURBS (Non Uniform Rational Beta Splines), and Hermite curves. We will discuss both plane curves and space curves here and also discussion of the tangent vector, blending functions, conic curves, re-parameterization, and continuity and composite curves.

Parametric Equations of a Curve

A parametric curve is one whose defining equations are given in terms of a single, common, independent variable called the parametric variable. We have already encountered parametric variables in earlier discussions of vectors, lines, and planes. Imagine a curve in three-dimensional space, each point on the curve has a unique set of coordinates: a specific x value, y value, and z value. Each coordinate is controlled by a separate parametric equation, whose general form looks like

$$x = x(u), \quad y = y(u), \quad z = z(u) \quad \dots\dots\dots(1)$$

where $x(u)$ stands for some as yet unspecified function in which u is the independent variable; for example, $x(u) = au^2 + bu + c$, and similarly for $y(u)$ and $z(u)$. It is important to understand that each of these is an independent expression. This will become clear as we discuss specific examples later.



The dependent variables are the $x, y,$ and z coordinates themselves, because their values depend on the value of the parametric variable u . Engineers and programmers who do geometric modeling usually prefer these kinds of expressions because the coordinates $x, y,$ and z are independent of each other, and each is defined by its own parametric equation.

Figure 1: point of a curve defined by a vector.

Each point on a curve is defined by a vector p (figure 1). The components of this vector are $x(u), y(u),$ and $z(u)$. We express this as

$$p = p(u) \quad \dots\dots\dots(2)$$

Which says that the vector p is a function of the parametric variable u .

There is a lot of information in equation 2. When we expand it into component form, it becomes

$$p(u) = [x(u) \quad y(u) \quad z(u)] \quad \dots\dots\dots(3)$$

The specific functions that define the vector components of p determine the shape of the curve. In fact, this is one way to define a curve – by simply choosing or designing these mathematical functions. There are only a few simple rules that we must follow: 1) define each component by a single, common parametric variable, and 2) make sure that each point on the curve corresponds to a unique value of the parametric variable. The last rule can be put another way: each value of the parametric variable must correspond to a unique point on the curve.

Plane Curves

To define plane curves, we use parametric functions that are second-degree polynomials:

$$\begin{aligned} x(u) &= a_x u^2 + b_x u + c_x \\ y(u) &= a_y u^2 + b_y u + c_y \quad \dots\dots\dots(4) \\ z(u) &= a_z u^2 + b_z u + c_z \end{aligned}$$

Where the a , b , and c terms are constant coefficients.

We can combine $x(u)$, $y(u)$, $z(u)$, and their respective coefficients into an equivalent, more concise, vector equation:

$$P(u) = au^2 + bu + c \quad \dots\dots\dots(5)$$

We allow the parametric variable to take on values only in the interval $0 \leq u \leq 1$. This ensures that the equation produces a bounded line segment. The coefficients a , b , c , in this equation are vectors, and each has three components; for example, $a = [a_x \ a_y \ a_z]$.

This curve has serious limitations. Although it can generate all the conic curves, or a close approximation to them, it cannot generate a curve with an inflection point, like an S-shaped curve, no matter what values we select for the coefficients a , b , c . To do this requires a cubic polynomial.

How do we define a specific plane curve, one that we can display, with defined end points, and a precise orientation in space? First, note in equation 4 or 5 that there are nine coefficients that we must determine: a_x , b_x , ..., c_z . If we know the two end points and the intermediate point.

End points and an intermediate point on the curve, then we now have nine quantities that we can express in terms of these coefficients (3 points x 3 coordinates each = 9 known quantities), and we can use these three points to define a unique curve (Figure 2). By applying some simple algebra to these relationships, we can rewrite Equation 5 in terms

of the three points. To one of the end points we assign $u = 0$, and to the other $u = 1$. To the intermediate point, we arbitrarily assign $u = 0.5$. We can write this points as

$$\begin{aligned} P_0 &= [x_0 \quad y_0 \quad z_0] \\ P_{0.5} &= [x_{0.5} \quad y_{0.5} \quad z_{0.5}] \\ P_1 &= [x_1 \quad y_1 \quad z_1] \end{aligned} \quad (6)$$

Where the subscripts indicate the value of the parametric variable at each point.

Now we solve equations 4 for the a_x, b_x, \dots, c_z coefficients in terms of these points. Thus, for x at $u = 0, u = 0.5$, and $u = 1$, we have

$$\begin{aligned} x_0 &= c_x \\ x_{0.5} &= 0.25a_x + 0.5b_x + c_x \\ x_1 &= a_x + b_x + c_x \end{aligned} \quad (7)$$

with similar equations for y , and z .

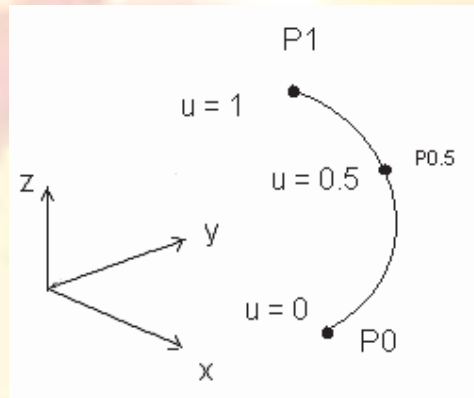


Figure 2: A plane curve defined by three points

Next we solve these three equations in three unknowns for a_x, b_x , and c_x , finding

$$\begin{aligned} a_x &= 2x_0 - 4x_{0.5} + 2x_1 \\ b_x &= -3x_0 + 4x_{0.5} - x_1 \\ c_x &= x_0 \end{aligned} \quad (8)$$

Substituting this result in to equation 4 yields

$$x(u) = (2x_0 - 4x_{0.5} + 2x_1) u^2 + (-3x_0 + 4x_{0.5} - x_1)u + x_0 \quad (9)$$

Again, there are equivalent expressions for $y(u)$ and $z(u)$. We rewrite equation 9 as follows:

$$x(u) = (2u^2 - 3u + 1) x_0 + (-4u^2 + 4u) x_{0.5} + (2u^2 - u) x_1 \quad (10)$$

Using this result and equivalent expressions for $y(u)$ and $z(u)$, we combine them into a single vector equation:

$$P(u) = (2u^2 - 3u + 1)P_0 + (-4u^2 + 4u)P_{0.5} + (2u^2 - u)P_1 \quad (11)$$

Equation 11 produces the same curve as Equation 5. The curve will always lie in a plane no matter what three points we choose. Furthermore, it is interesting to note that the point $P_{0.5}$ which is on the curve at $u = 0.5$, is not necessarily half way along the length of the curve between P_0 and P_1 . We can show this quite convincingly by choosing three points to define a curve such that two of them are relatively close together (figure 3). In fact, if we assign a different value to the parametric variable for the intermediate point, then we obtain different values for the coefficients in equations 8. This, in turn, means that a different curve is produced, although it passes through the same three points.

Equation 5 is the algebraic form and equation 11 is the geometric form. Each of these equations can be written more compactly with matrices. Compactness is not the only advantage to matrix notation. Once a curve is defined in matrix form, we can use the full power of matrix algebra to solve many geometry problems.

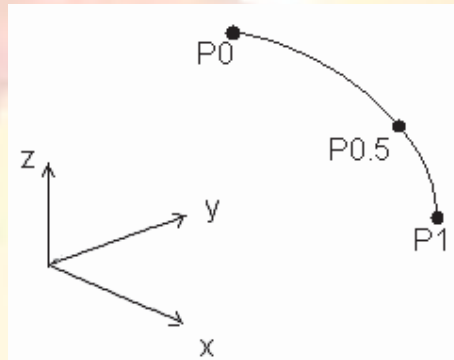


Figure 3: Curve defined by three non-uniformly spaced points.

So now we rewrite equation 5 using the following substitutions:

$$\begin{bmatrix} u^2 & u & 1 \end{bmatrix} \begin{bmatrix} a \\ b \\ c \end{bmatrix} = au^2 + bu + c \quad \dots\dots\dots(12)$$

$$U = \begin{bmatrix} u^2 & u & 1 \end{bmatrix} \quad \dots\dots\dots(13)$$

$$A = \begin{bmatrix} a & b & c \end{bmatrix}^T \quad \dots\dots\dots(14)$$

And finally we obtain

$$p(u) = UA \quad \dots\dots\dots(15)$$

Remember that A is really a matrix of vectors, so that

$$A = \begin{bmatrix} a \\ b \\ c \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \end{bmatrix} \quad \dots\dots\dots(16)$$

The nine terms on the right are called the algebraic coefficients.

Next, we convert equation 11 into matrix form. The right-hand side looks like the product of two matrices: $[(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)]$ and $[p_0 \quad p_{0.5} \quad p_1]$. This means that

$$p(u) = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} \quad \dots\dots\dots(17)$$

Using the following substitutions:

$$F = [(2u^2 - 3u + 1) \quad (-4u^2 + 4u) \quad (2u^2 - u)] \quad \dots\dots\dots(18)$$

and

$$p = \begin{bmatrix} p_0 \\ p_{0.5} \\ p_1 \end{bmatrix} = \begin{bmatrix} x_0 & y_0 & z_0 \\ x_{0.5} & y_{0.5} & z_{0.5} \\ x_1 & y_1 & z_1 \end{bmatrix} \quad \dots\dots\dots(19)$$

where P is the control point matrix and the nine terms on the right are its elements or the geometric coefficients, we can now write

$$p(u) = FP \quad \dots\dots\dots(20)$$

This is the matrix version of the geometric form.

Because it is the same curve in algebraic form, $p(u)=UA$, or geometric form, $p(u)=FP$, we can write

$$FP = UA \quad \dots\dots\dots(21)$$

The F matrix is itself the product of two other matrices:

$$F = [u^2 \quad u \quad 1] \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(22)$$

The matrix on the left we recognize as U, and we can denote the other matrix as

$$M = \begin{bmatrix} 2 & -4 & 2 \\ -3 & 4 & -1 \\ 1 & 0 & 0 \end{bmatrix} \dots\dots\dots(23)$$

This means that

$$F = UM \dots\dots\dots(24)$$

Using this we substitute appropriately to find

$$UMP = UA \dots\dots\dots(25)$$

Pre-multiplying each side of this equation by U^{-1} yields

$$MP = A \dots\dots\dots(26)$$

This expresses a simple relationship between the algebraic and geometric coefficients

$$A = MP \dots\dots\dots(27)$$

Or

$$P = M^{-1}A \dots\dots\dots(28)$$

The matrix M is called a basis transformation matrix, and F is called a blending function matrix.

Lecture No.36 Space Curves

A space curve is not confined to a plane. It is free to twist through space. To define a space curve we must use parametric functions that are cubic polynomials. For $x(u)$ we write

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \quad (1)$$

With similar expressions for $y(u)$ and $z(u)$. Again the a , b , c and d terms are constant coefficients. As we did with Equation for a plane curve, we combine the $x(u)$, $y(u)$, and $z(u)$ expressions into a single vector equation :

$$p(u) = au^3 + bu^2 + cu + d \quad (2)$$

If $\mathbf{a} = \mathbf{0}$, then this equation is identical to Equation discussed in plane curves

To define a specific curve in space, we use the same approach as we did for a plane curve. This time, though, there are 12 coefficients to be determined. We specify four points through which we want the curve to pass, which provides all the information we need to determine \mathbf{a} , \mathbf{b} , \mathbf{c} , and \mathbf{d} . but which four points? Two are obvious: $p(0)$ and $p(1)$, the end points at $u=0$ and $u=1$. For various reasons beyond our scope, it turns out to be advantageous to use two intermediate points that we assign parametric values of $u = \frac{1}{3}$ and $u = \frac{2}{3}$, or $p\left(\frac{1}{3}\right)$ and $p\left(\frac{2}{3}\right)$. So we now have the four points we need:

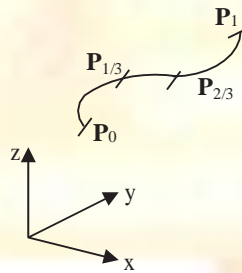


Figure 1 Four points define a cubic space curve

$p(0)$, $p\left(\frac{1}{3}\right)$ and $p\left(\frac{2}{3}\right)$ and $p(1)$, which we can rewrite as the more convenient p_1 , p_2 , p_3 , and p_4 (Figure 1).

Substituting each of the values of the parametric variable ($u=0, 1/3, 2/3, 1$) into Equation 1, we obtain the following four equations in four unknowns:

$$\begin{aligned} x_1 &= d_x \\ x_2 &= \frac{1}{27}a_x + \frac{1}{9}b_x + \frac{1}{3}c_x + d_x \\ x_3 &= \frac{8}{27}a_x + \frac{4}{9}b_x + \frac{2}{3}c_x + d_x \\ x_4 &= a_x + b_x + c_x + d_x \end{aligned} \quad (3)$$

$$x_4 = a_x + b_x + c_x + d_x$$

Now we can express a_x , b_x , c_x and d_x in terms of x_1 , x_2 , x_3 , and x_4 . After doing the necessary algebra, we obtain

$$\begin{aligned} a_x &= -\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4 \\ b_x &= 9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4 \\ c_x &= -\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4 \\ d_x &= x_1 \end{aligned} \quad (4)$$

We substitute these results into Equation 1, producing

$$\begin{aligned} x(u) &= \left(-\frac{9}{2}x_1 + \frac{27}{2}x_2 - \frac{27}{2}x_3 + \frac{9}{2}x_4 \right) u^3 + \left(9x_1 - \frac{45}{2}x_2 + 18x_3 - \frac{9}{2}x_4 \right) u^2 \\ &+ \left(-\frac{11}{2}x_1 + 9x_2 - \frac{9}{2}x_3 + x_4 \right) u + x_1 \end{aligned} \quad (5)$$

All this looks a bit messy right now, but we can put it into a neat, much more compact form. We begin by rewriting Equation 5 as follows:

$$\begin{aligned} x(u) &= \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right) x_1 + \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right) x_2 \\ &+ \left(-\frac{27}{2}u^3 + 18u^2 - 9u \right) x_3 + \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right) x_4 \end{aligned} \quad (6)$$

Using equivalent expressions for $y(u)$ and $z(u)$, we can summarize them as a single vector equation:

$$\begin{aligned} p(u) &= \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right) p_1 + \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right) p_2 \\ &+ \left(-\frac{27}{2}u^3 + 18u^2 - 9u \right) p_3 + \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right) p_4 \end{aligned} \quad (7)$$

This means that, given four point assigned successive values of u (in this case at $u=0, 1/3, 2/3, 1$), equation 7 produces a curve that starts at p_1 , passes through p_2 and p_3 , and ends at p_4 .

Now let's take one more step toward a more compact notation. Using the four parametric functions appearing in Equation 7, we define a new matrix, $G = [G_1 \ G_2 \ G_3 \ G_4]$, where

$$\begin{aligned} G_1 &= \left(-\frac{9}{2}u^3 + 9u^2 - \frac{11}{2}u + 1 \right) & G_2 &= \left(\frac{27}{2}u^3 - \frac{45}{2}u^2 + 9u \right) \\ G_3 &= \left(-\frac{27}{2}u^3 + 18u^2 - 9u \right) & G_4 &= \left(\frac{9}{2}u^3 - \frac{9}{2}u^2 + u \right) \end{aligned} \quad (8)$$

And then define a matrix P containing the control points, $P = [P_1 \ P_2 \ P_3 \ P_4]$, so that

$$P(u) = GP \quad (9)$$

The matrix G is the product of two other matrices, U and N :

$$G = UN \quad (10)$$

Where $U = [u^3 \ u^2 \ u \ 1]^T$ and

$$N = \begin{bmatrix} -\frac{9}{2} & -\frac{27}{2} & \frac{27}{2} & 9 \\ 9 & -\frac{45}{2} & -\frac{9}{2} & 1 \\ -\frac{11}{2} & 9 & -\frac{9}{2} & 1 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (11)$$

(Note that N is another example of a basis transformation matrix.)

Now we let

$$A = \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} a_x & a_y & a_z \\ b_x & b_y & b_z \\ c_x & c_y & c_z \\ d_x & d_y & d_z \end{bmatrix} \quad (12)$$

Using matrices, Equation 2 becomes

$$P(u) = UA \quad (13)$$

Which looks a lot like Equation for a plane curve, except that we have defined new U and A matrices? In fact, Equation is a special case of the formulation for a space curve.

To convert the information in the A matrix into that required for the P matrix, we do some simple matrix algebra, using Equations 9, 10 and 13. First we have

$$GP = UNP \quad (14)$$

And then

$$UA = UNP \quad (15)$$

Or more simply

$$A = NP \quad (16)$$

Lecture No.37 The Tangent Vector

Another way to define a space curve does not use intermediate points. It uses the tangents at each end of the curve, instead. Every point on a curve has a straight line associated with it called the tangent line, which is related to the first derivation of the Parametric functions $x(u)$, $y(u)$, and $z(u)$, such as those given by Equation 2 of previous lecture. Thus

$$\frac{d}{du}x(u), \quad \frac{d}{du}y(u), \quad \text{and} \quad \frac{d}{du}z(u) \quad (1)$$

From elementary calculus, we can compute, for example,

$$\frac{dy}{du} = \frac{dy(u)/du}{dx(u)/du} \quad (2)$$

We can treat $dx(u)/du, dy(u)/du, \text{ and } dz(u)/du$ as components of a vector along the tangent line to the curve. We call this the *tangent vector*, and define it as

$$P^u(u) = \left[\frac{d}{du}x(u)i \quad \frac{d}{du}y(u)j \quad \frac{d}{du}z(u)k \right] \quad (3)$$

Or more simply as

$$P^u = [x^u \quad y^u \quad z^u] \quad (4)$$

(Here the superscript u indicates the first derivative operation with respect to the independent variable u). This is a very powerful idea, and we will now see how to use it to define a curve.

In the last section, we discussed how to define a curve by specifying four points. Now we have another way to define a curve. We will still use the two end points, but instead of two intermediate points, we will use the tangent vectors at each end to supply the information we need to define a curve. By manipulating these tangent vectors, we can control the slope at each end. The set of vectors $p_0, p_1, p_0^u, \text{ and } p_1^u$ are called the *boundary conditions*. This method itself is called the *cubic Hermite interpolation*, after C. Hermite (1822-1901) the French mathematician who made significant contributions to our understanding of cubic and quadratic polynomials.

We differentiate to obtain the x component of the tangent vector:

$$\frac{d}{du} dx(u) = x^u = 3a_x u^2 + 2b_x u + c_x \quad (5)$$

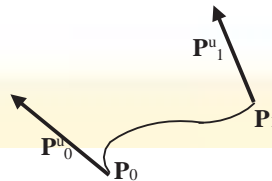


Figure 1 Defining a curve using end points and tangent vectors.

$$x(u) = a_x u^3 + b_x u^2 + c_x u + d_x \quad (1A)$$

Evaluating (1A) and Equation 5 at $u = 0, u = 1$, yields

$$\begin{aligned}x_0 &= d_x \\x_1 &= a_x + b_x + c_x + d_x \\x''_0 &= c_x \\x''_1 &= 3a_x + 2b_x + c_x\end{aligned}\quad (6)$$

Using these four equations in four unknowns, we solve for a_x , b_x , c_x and d_x in terms of the boundary conditions

$$\begin{aligned}a_x &= 2(x_0 - x_1) + x''_0 + x''_1 \\b_x &= 3(-x_0 + x_1) - 2x''_0 - x''_1 \\c_x &= x''_0 \\d_x &= x_0\end{aligned}\quad (7)$$

Substituting the result into Equation (1A), yields

$$x(u) = (2x_0 - 2x_1 + x''_0 + x''_1)u^3 + (-3x_0 + 3x_1 - 2x''_0 - x''_1)u^2 + x''_0 + x_0 \quad (8)$$

Rearranging terms we can rewrite this as

$$x(u) = (2u^3 - 3u^2 + 1)x_0 + (-2u^3 + 3u^2)x_1 + (u^3 - 2u^2 + u)x''_0 + (u^3 - u^2)x''_1 \quad (9)$$

Because $y(u)$ and $z(u)$ have equivalent forms, we can include them by rewriting Equation 9 in vector form:

$$p(u) = (2u^3 - 3u^2 + 1)p_0 + (-2u^3 + 3u^2)p_1 + (u^3 - 2u^2 + u)p''_0 + (u^3 - u^2)p''_1 \quad (10)$$

To express Equation 10 in matrix notation, we first define a blending function matrix

$F = [F_1 \ F_2 \ F_3 \ F_4]$, where

$$\begin{aligned}F_1 &= 2u^3 - 3u^2 + 1 \\F_2 &= -2u^3 + 3u^2 \\F_3 &= u^3 - 2u^2 + u \\F_4 &= u^3 - u^2\end{aligned}\quad (11)$$

These matrix elements are the polynomial coefficients of the vectors which we rewrite as

$$p(u) = F_1 p_0 + F_2 p_1 + F_3 p''_0 + F_4 p''_1 \quad (12)$$

If we assemble the vectors representing the boundary conditions into a matrix \mathbf{B} ,

$$B = \begin{bmatrix} p_0 & p_1 & p''_0 & p''_1 \end{bmatrix} \quad (13)$$

Then

$$p(u) = FB \quad (14)$$

Here again we write the matrix \mathbf{F} as the product of two matrices, \mathbf{U} and \mathbf{M} , so that

$$F = UM \quad (15)$$

where

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad (16)$$

and

$$M = \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (17)$$

Rewriting Equation 14 using these substitutions, we obtain

$$P(u) = UMB \quad (18)$$

It is easy to show that the relationship between the algebraic and geometric coefficients for a space curve. Since

$$P(u) = UA \quad (19)$$

the relationship between **A** and **B** is, again,

$$A = MB \quad (20)$$

Consider the four vectors that make up the boundary condition matrix. There is nothing extraordinary about the vectors defining the end points, but what about the two tangent vector? A tangent vector certainly defines the slope at one end of the curve, but a vector has characteristics of both direction and magnitude. All we need to specify the slope is a unit tangent vector at each end, say t_0 and t_1 . But p_0 , p_1 , t_0 , and t_1 supply only 10 of the 12 pieces of information needed to completely determine the curve. So the magnitude of the tangent vector is also necessary and contributes to the shape of the curve. In fact, we can write p_0'' and p_1'' as:

$$p_0'' = m_0 t_0 \quad (21)$$

And

$$p_1'' = m_1 t_1 \quad (22)$$

Clearly, m_0 , and m_1 are the magnitudes of p_0'' and p_1'' .

Using these relationships, we modify Equation 10 as follows:

$$p(u) = (2u^3 - 3u^2 + 1)p_0 + (-2u^3 + 3u^2)p_1 + (u^3 - 2u^2 + u)m_0 t_0 + (u^3 - u^2)m_1 t_1 \quad (23)$$

Now we can experiment with a curve (Figure 2). Let's hold p_0 , p_1 , t_0 , and t_1 constants and see what happens to the shape of the curve as we vary m_0 and m_1 . For simplicity we will consider a curve in the x, y plane. This means that z_0 , z_1 , z_0'' and z_1'' are all equal to zero.

The **B** matrix for the curve drawn with the bold line (and with $m_0 = m_1 = 1$) is

$$\mathbf{B} = \begin{bmatrix} P_0 \\ P_1 \\ m_0 t_0 \\ m_1 t_1 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 \\ 1 & 0 & 0 \\ 0.707 & 0.707 & 0 \\ 0.707 & -0.707 & 0 \end{bmatrix} \quad (24)$$

Carefully consider this array of 12 elements; they uniquely define the curve. By changing either m_0 or m_1 , or both, we can change the shape of the curve. But it is a restricted kind of change because not only do the end points remain fixed, but the end slopes are also unchanged!

The three curves drawn with light lines in Figure 3 show the effects of varying m_0 and m_1 . This is a very powerful tool for designing curves, making it possible to join up end-to-end many curves in a smooth way and still exert some control over the interior shape of each individual curve. For example, as we increase the value of m_0 while holding m_1 fixed, the curve seems to be pushed toward \mathbf{p}_1 . Keeping m_0 and m_1 equal but increasing their value increases the maximum deflection of the curve from the x -axis and increases the curvature at the maximum. (Under some conditions, not necessarily desirable, we can force a loop to form).

Lecture No.38 Bezier Curves

The Bezier curve is an important part of almost every computer-graphics illustration program and computer-aided design system in use today. It is used in many ways, from designing the curves and surfaces of automobiles to defining the shape of letters in type fonts. And because it is numerically the most stable of all the polynomial-based curves used in these applications, the Bezier curve is the ideal standard for representing the more complex piecewise polynomial curves.

In the early 1960s, Peter Bezier began looking for a better way to define curves and surfaces one that would be useful to a design engineer. He was familiar with the work of Ferguson and Coons and their parametric cubic curves and bicubic surfaces. However, these did not offer an intuitive way to alter and control shape. The results of Bezier's research led to the curves and surfaces that bear his name and became part of the UNISURF system. The French automobile manufacturer, Renault used UNISURF to design the sculptured surfaces of many of its products.

A Geometric Construction

We can draw a Bezier curve using a simple recursive geometric construction. Let's begin by constructing a second-degree curve. We select three points A, B, C so that line AB is a tangent to the curve at A, and BC is tangent at C. The curve begins at A and ends at C. For any ratio, u_i , we construct points D and E so that



Figure 1: Geometric Construction of a second-degree Bezier curve.

$$\frac{AD}{AB} = \frac{BE}{BC} = u_i \quad (1)$$

On DE we construct F so that $DF/DE = u_j$. Point F is on the curve. Repeating this process for other values of u_i , we produce a series of points on a Bezier curve. Note that we must be consistent in the order in which we sub-divide AB and BC.

To define this curve in a coordinate system, let point A = (x_A, y_A) , B = (x_B, y_B) . Then coordinates of points D and E for some value of u_i are

$$\begin{aligned} x_D &= x_A + u_i(x_B - x_A) \\ y_D &= y_A + u_i(y_B - y_A) \end{aligned} \quad (2)$$

And

$$\begin{aligned}x_E &= x_B + u_i(x_C - x_B) \\y_E &= y_B + u_i(y_C - y_B)\end{aligned}\quad (3)$$

The coordinates of point F for some value of u_i are

$$\begin{aligned}x_F &= x_D + u_i(x_E - x_D) \\y_F &= y_D + u_i(y_E - y_D)\end{aligned}\quad (4)$$

To obtain x_F and y_F in terms of the coordinates of points A, B and C, for any value of u_i in the unit interval, we substitute appropriately from Equation 2 and 3 into equations 4 of plane curve. After rearranging terms to simplify, we find

$$\begin{aligned}x_F &= (1-u_i)^2 x_A + 2u_i(1-u_i)x_B + u_i^2 x_C \\y_F &= (1-u_i)^2 y_A + 2u_i(1-u_i)y_B + u_i^2 y_C\end{aligned}\quad (5)$$

We generalize this set of equations for any point on the curve using the following substitutions:

$$\begin{aligned}x(u) &= x_F \\y(u) &= y_F\end{aligned}\quad (6)$$

And we let

$$\begin{aligned}x_0 &= x_A & x_1 &= x_B & x_2 &= x_C \\y_0 &= y_A & y_1 &= y_B & y_2 &= y_C\end{aligned}\quad (7)$$

Now we can rewrite Equation 5 as

$$\begin{aligned}x(u) &= (1-u)^2 x_0 + 2u(1-u)x_1 + u^2 x_2 \\y(u) &= (1-u)^2 y_0 + 2u(1-u)y_1 + u^2 y_2\end{aligned}\quad (8)$$

This is the set of second-degree equations for the coordinates of points on Bezier curve based on our construction.

We express this construction process and Equations 8 in terms of vectors with the following substitutions. Let the vector p_0 represent point A, p_1 point B, and p_2 point C. From vector geometry we have $D = p_0 + u(p_1 - p_0)$ and $E = p_1 + u(p_2 - p_1)$ if we let $F = p_0 + u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 + u(p_1 - p_0)]$ we see that

$$p(u) = p_0 + u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 + u(p_1 - p_0)]\quad (9)$$

We rearrange terms to obtain a more compact vector equation of a second degree Bezier curve:

$$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2\quad (10)$$

The ratio u is the parametric variable. Later we will see that this equation is an example of a *Bernstein polynomial*. Note that the curve will always lie in the plane containing the three control points, but the points do not necessarily lie in the xy plane.

Similar constructions apply to Bezier curves of any degree. In fact the degree of a Bezier curve is equal to $n-1$, where n is the number of control points.

Figure 2 shows the construction of point on a cubic Bezier curve, which requires four control points A, B, C, and D to define it. The curve begins at point A tangent to the, AB, and ends at D and tangent to CD. We construct points E, F, and G so that

$$\frac{AE}{AB} = \frac{BF}{BC} = \frac{CG}{CD} = u_i \quad (11)$$

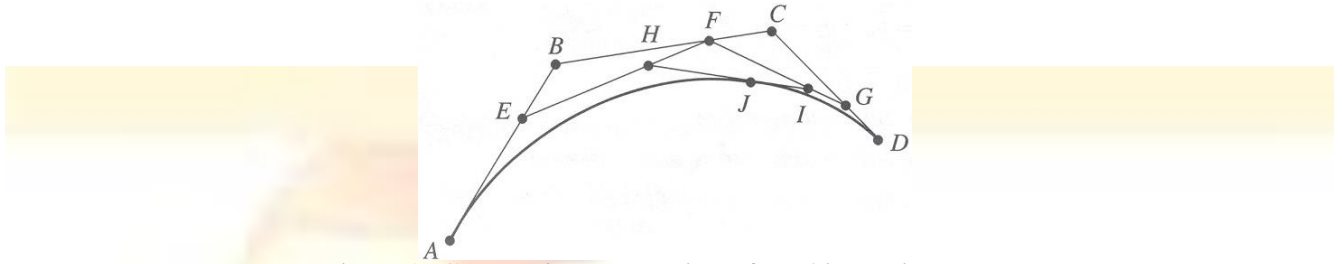


Figure 2: Geometric construction of a cubic Bezier curve

On EF and FG we locate H and I, respectively, so that

$$\frac{EH}{EF} = \frac{FI}{FG} = u_i \quad (12)$$

Finally on HI we locate J so that

$$\frac{HJ}{HI} = u_i \quad (13)$$

We can make no more subdivisions, which means that point J is on the curve. If we continue this process for a sequence of points, then their locus defines the curve. If points A, B, C, and D are represented by the vectors $\mathbf{p}_0, \mathbf{p}_1, \mathbf{p}_2, \mathbf{p}_3$ respectively, then expressing the construction of the intermediate points E, F, G, H, and I in terms of these vectors to produce point J, or $P(u)$, yields

$$\begin{aligned} p(u) = & p_0 + u(p_1 - p_0) + u\{p_1 + u(p_2 - p_1) - p_0 + u(p_1 - p_0)\} \\ & + u\{p_1 + u(p_2 - p_1) + u[p_2 + u(p_3 - p_2) - p_1 + u(p_2 - p_0)] - p_0 \\ & - u(p_1 - p_0) + u[p_1 + u(p_2 - p_1) - p_0 - u(p_1 - p_0)]\} \end{aligned} \quad (14)$$

This awkward expression simplifies nicely to

$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3 \quad (15)$$

Of course this construction, of a cubic curve with its four control points is done in the plane of the paper. However, the cubic polynomial allows a curve that is nonplanar; that is, it can represent a curve that twists in space.

The geometric construction of a Bezier curve shows how the control points influence its shape. The curve begins on the first point and ends on the last point. It is tangent to the lines connecting the first two points and the last two points. The curve is always contained within the *convex hull* of the control points.

No one spends time constructing and plotting the points of a Bezier curve by hand, of course. A computer does a much faster and more accurate job. However, it is worth doing several curves this way for insight into the characteristics of Bezier curves.

An Algebraic Definition

Bezier began with the idea that any point $p(u)$ on a curve segment should be given by an equation such as the following:

$$p(u) = \sum_{i=0}^n p_i f_i(u) \quad (16)$$

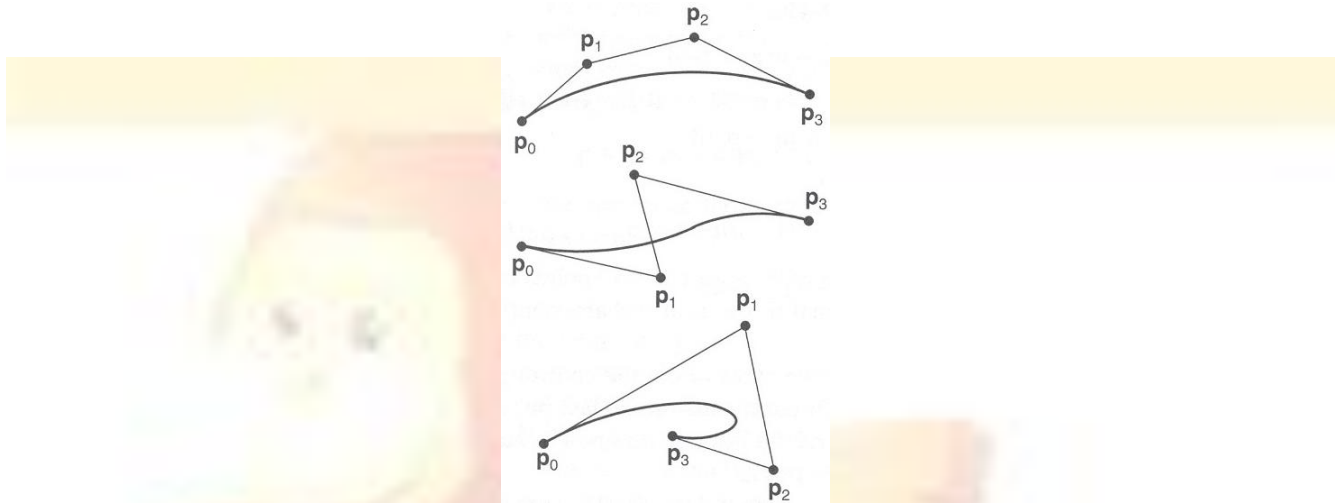


Figure 3: Bezier curves and their control points

Equation 16 is a compact way to express the sum of several similar terms because what it says is this:

$$p(u) = p_0 f_0(u) + p_1 f_1(u) + \dots + p_n f_n(u) \quad (17)$$

Of which Equation 10 and 15 are specific examples, for $n=2$ and $n=3$, respectively. The $n+1$ functions, that is the $f_i(u)$ must produce a curve that has certain well-defined characteristics. Here are some of the most important ones:

1. The curve must start on the first control point, \mathbf{p}_0 , and end on the last, \mathbf{p}_n . Mathematically, we say that the functions must interpolate these two points.
2. The curve must be tangent to the line given by $\mathbf{p}_1 - \mathbf{p}_0$ at \mathbf{p}_0 and $\mathbf{p}_n - \mathbf{p}_{n-1}$ at \mathbf{p}_n .
3. The functions $f_i(u)$ must be symmetric with respect to u and $(1-u)$. This lets us reverse the sequence of control points without changing the shape of the curve.

Other characteristics can be found in more advanced works on the subject.

Figure

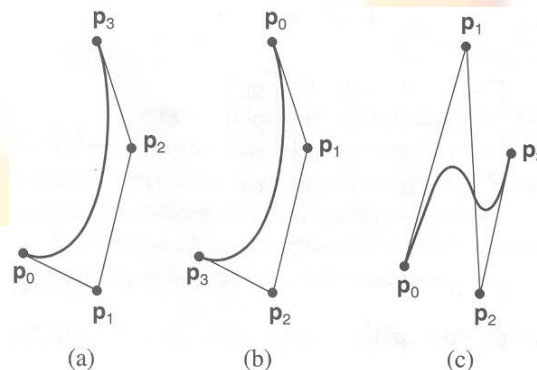


Figure 4: Three different sequences of four control points

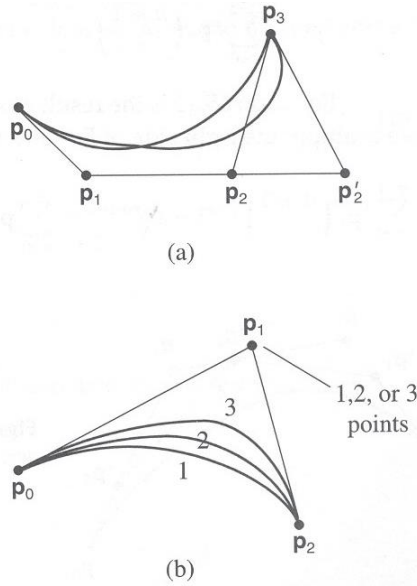


Figure 5: Modifying the shape of a Bezier curve.

A family of functions called *Bernstein polynomials* satisfies these requirements. They are the *basis functions* of the Bezier curve (Other curves, such as the NURBS curves, use different, but related, basis functions). We rewrite Equation 16 using them so that

$$p(u) = \sum_{i=0}^n p_i B_{i,n}(u) \quad (18)$$

Where the basis functions are

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i} \quad (19)$$

$$\binom{n}{i} = \frac{n!}{i!(n-i)!} \quad (20)$$

The symbol ! is the factorial operator. For example, $3! = 3 \times 2 \times 1$, $5! = 5 \times 4 \times 3 \times 2 \times 1$, so forth. We use the following conventions when evaluating Equation 20; If i and u equal zero, then $u^i = 1$ and $0! = 1$. We see that for $n+1$ control points, the basis functions produce an n th-degree polynomial.

Expanding Equation 18 for a second degree Bezier curve (When $n=2$ and there are three control points) produces

$$p(u) = p_0 B_{0,2}(u) + p_1 B_{1,2}(u) + p_2 B_{2,2}(u) \quad (21)$$

From Equation 20, we find

$$B_{0,2}(u) = (1-u)^2 \quad (22)$$

$$B_{1,2}(u) = 2u(1-u) \quad (23)$$

$$B_{2,2}(u) = u^2 \quad (24)$$

These are the basis functions for a second-degree Bezier curve.

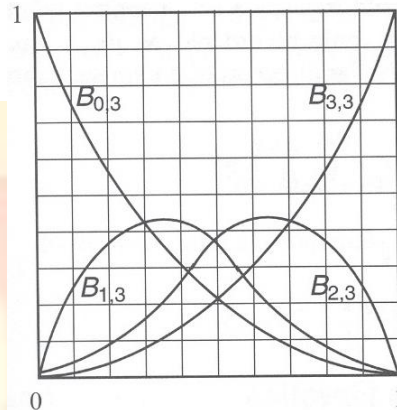


Figure 6: Bezier curve basis functions

Substituting them into Equation 21 and rearranging terms, we find

$$p(u) = (1-u)^2 p_0 + 2u(1-u)p_1 + u^2 p_2 \quad (25)$$

This is the same expression we found from the geometric construction, Equation 10. The variable u is now called the parametric variable.

Now, let's expand Equation 18 for a cubic Bezier curve, where $n=3$:

$$p(u) = p_0 B_{0,3}(u) + p_1 B_{1,3}(u) + p_2 B_{2,3}(u) + p_3 B_{3,3}(u) \quad (26)$$

And from Equation 20 we find

$$B_{0,3}(u) = (1-u)^3 \quad (27)$$

$$B_{1,3}(u) = 3u(1-u)^2 \quad (28)$$

$$B_{2,3}(u) = 3u^2(1-u) \quad (29)$$

$$B_{3,3}(u) = u^3 \quad (30)$$

Substituting these into Equation 26 and rearranging terms produces

$$p(u) = (1-u)^3 p_0 + 3u(1-u)^2 p_1 + 3u^2(1-u)p_2 + u^3 p_3 \quad (31)$$

Bezier curve equations are well suited for expression in matrix form. We can expand the cubic parametric functions and rewrite Equation 31 as

$$p(u) = \begin{bmatrix} (1-3u+3u^2-u^3) \\ (3u-6u^2+3u^3) \\ (3u^2-3u^3) \\ u^3 \end{bmatrix}^T \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad (32)$$

or as

$$p(u) = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \begin{bmatrix} -1 & -3 & -3 & 0 \\ 3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ p_3 \end{bmatrix} \quad (33)$$

If we let

$$U = \begin{bmatrix} u^3 & u^2 & u & 1 \end{bmatrix} \quad (34)$$

$$P = \begin{bmatrix} p_0 & p_1 & p_2 & p_3 \end{bmatrix}^T \quad (35)$$

and

$$M = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} \quad (36)$$

Then we can write Equation 33 even more compactly as

$$p(u) = UMP \quad (37)$$

Note that the composition of the matrices **U**, **M**, and **P** varies according to the number of control points (that is, the degree of the Bernstein polynomial basis functions).

Lecture No.39 Building Polygonal Models of Surfaces

Constructing polygonal approximations to surfaces is an art, and there is no substitute for experience. This section, however, lists a few pointers that might make it a bit easier to get started.

- Keep polygon orientations consistent. Make sure that when viewed from the outside, all the polygons on the surface are oriented in the same direction (all clockwise or all counterclockwise). Consistent orientation is important for polygon culling and two-sided lighting. Try to get this right the first time, since it's excruciatingly painful to fix the problem later. (If you use `glScale*()` to reflect geometry around some axis of symmetry, you might change the orientation with `glFrontFace()` to keep the orientations consistent.)
- When you subdivide a surface, watch out for any nontriangular polygons. The three vertices of a triangle are guaranteed to lie on a plane; any polygon with four or more vertices might not. Nonplanar polygons can be viewed from some orientation such that the edges cross each other, and OpenGL might not render such polygons correctly.
- There's always a trade-off between the display speed and the quality of the image. If you subdivide a surface into a small number of polygons, it renders quickly but might have a jagged appearance; if you subdivide it into millions of tiny polygons, it probably looks good but might take a long time to render. Ideally, you can provide a parameter to the subdivision routines that indicates how fine a subdivision you want, and if the object is farther from the eye, you can use a coarser subdivision. Also, when you subdivide, use large polygons where the surface is relatively flat, and small polygons in regions of high curvature.
- For high-quality images, it's a good idea to subdivide more on the silhouette edges than in the interior. If the surface is to be rotated relative to the eye, this is tougher to do, since the silhouette edges keep moving. Silhouette edges occur where the normal vectors are perpendicular to the vector from the surface to the viewpoint - that is, when their vector dot product is zero. Your subdivision algorithm might choose to subdivide more if this dot product is near zero.
- Try to avoid T-intersections in your models (see Figure 1). As shown, there's no guarantee that the line segments AB and BC lie on exactly the same pixels as the segment AC. Sometimes they do, and sometimes they don't, depending on the transformations and orientation. This can cause cracks to appear intermittently in the surface.

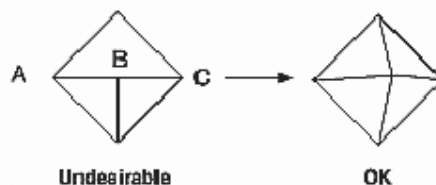


Figure 1 : Modifying an Undesirable T-intersection

- If you're constructing a closed surface, make sure to use exactly the same numbers for coordinates at the beginning and end of a closed loop, or you can get gaps and cracks due to numerical round-off. Here's a two-dimensional example of bad code:

```

#define PI 3.14159265
#define EDGES 30
/* draw a circle */
glBegin(GL_LINE_STRIP);

for (i = 0; i <= EDGES; i++)
    glVertex2f(cos((2*PI*i)/EDGES), sin((2*PI*i)/EDGES));
glEnd();

```

The edges meet exactly only if your machine manages to calculate the sine and cosine of 0 and of $(2*PI*EDGES/EDGES)$ and gets exactly the same values. If you trust the floating-point unit on your machine to do this right, the authors have a bridge they'd like to sell you. To correct the code, make sure that when $i == EDGES$, you use 0 for the sine and cosine, not $2*PI*EDGES/EDGES$. (Or simpler still, use `GL_LINE_LOOP` instead of `GL_LINE_STRIP`, and change the loop termination condition to $i < EDGES$.)

An Example: Building an Icosahedron

To illustrate some of the considerations that arise in approximating a surface, let's look at some example code sequences. This code concerns the vertices of a regular icosahedron (which is a Platonic solid composed of twenty faces that span twelve vertices, each face of which is an equilateral triangle). An icosahedron can be considered a rough approximation for a sphere. Example 1 defines the vertices and triangles making up an icosahedron and then draws the icosahedron.

Example 1 : Drawing an Icosahedron

```

#define X .525731112119133606
#define Z .850650808352039932
static GLfloat vdata[12][3] = {
    {-X, 0.0, Z}, {X, 0.0, Z}, {-X, 0.0, -Z}, {X, 0.0, -Z},
    {0.0, Z, X}, {0.0, Z, -X}, {0.0, -Z, X}, {0.0, -Z, -X},
    {Z, X, 0.0}, {-Z, X, 0.0}, {Z, -X, 0.0}, {-Z, -X, 0.0}
};
static GLuint tindices[20][3] = {
    {0,4,1}, {0,9,4}, {9,5,4}, {4,5,8}, {4,8,1},
    {8,10,1}, {8,3,10}, {5,3,8}, {5,2,3}, {2,7,3},
    {7,10,3}, {7,6,10}, {7,11,6}, {11,0,6}, {0,1,6},
    {6,1,10}, {9,0,11}, {9,11,2}, {9,2,5}, {7,2,11}
};

int i;
glBegin(GL_TRIANGLES);
    for (i = 0; i < 20; i++) {
        /* color information here */
        glVertex3fv(&vdata[tindices[i][0]][0]);
        glVertex3fv(&vdata[tindices[i][1]][0]);
        glVertex3fv(&vdata[tindices[i][2]][0]);
    }
glEnd();

```

The strange numbers X and Z are chosen so that the distance from the origin to any of the vertices of the icosahedron is 1.0. The coordinates of the twelve vertices are given in the array `vdata[][]`, where the zeroth vertex is `{ - &Xgr; , 0.0, &Zgr; }`, the first is `{ X, 0.0, Z }`, and so on. The array `tindices[][]` tells how to link the vertices to make triangles. For example, the first triangle is made from the zeroth, fourth, and first vertex. If you take the vertices for triangles in the order given, all the triangles have the same orientation.

The line that mentions color information should be replaced by a command that sets the color of the *i*th face. If no code appears here, all faces are drawn in the same color, and it'll be impossible to discern the three-dimensional quality of the object. An alternative to explicitly specifying colors is to define surface normals and use lighting, as described in the next subsection.

Note: In all the examples described in this section, unless the surface is to be drawn only once, you should probably save the calculated vertex and normal coordinates so that the calculations don't need to be repeated each time that the surface is drawn. This can be done using your own data structures or by constructing display lists.

Calculating Normal Vectors for a Surface

If a surface is to be lit, you need to supply the vector normal to the surface. Calculating the normalized cross product of two vectors on that surface provides normal vector. With the flat surfaces of an icosahedron, all three vertices defining a surface have the same normal vector. In this case, the normal needs to be specified only once for each set of three vertices. icosahedron.

Example 2 : Generating Normal Vectors for a Surface

```
GLfloat d1[3], d2[3], norm[3];
for (j = 0; j < 3; j++) {
    d1[j] = vdata[tindices[i][0]][j] - vdata[tindices[i][1]][j];
    d2[j] = vdata[tindices[i][1]][j] - vdata[tindices[i][2]][j];
}
normcrossprod(d1, d2, norm);
glNormal3fv(norm);
```

The function `normcrossprod()` produces the normalized cross product of two vectors, as shown in Example 3.

Example 3 : Calculating the Normalized Cross Product of Two Vectors

```
void normalize(float v[3]) {
    GLfloat d = sqrt(v[0]*v[0]+v[1]*v[1]+v[2]*v[2]);
    if (d == 0.0) {
        error("zero length vector");
        return;
    }
    v[0] /= d; v[1] /= d; v[2] /= d;
}
```

```

void normcrossprod(float v1[3], float v2[3], float out[3])
{
    GLint i, j;
    GLfloat length;
    out[0] = v1[1]*v2[2] - v1[2]*v2[1];
    out[1] = v1[2]*v2[0] - v1[0]*v2[2];
    out[2] = v1[0]*v2[1] - v1[1]*v2[0];
    normalize(out);
}

```

If you're using an icosahedron as an approximation for a shaded sphere, you'll want to use normal vectors that are perpendicular to the true surface of the sphere, rather than being perpendicular to the faces. For a sphere, the normal vectors are simple; each point in the same direction as the vector from the origin to the corresponding vertex. Since the icosahedron vertex data is for an icosahedron of radius 1, the normal and vertex data is identical. Here is the code that would draw an icosahedral approximation of a smoothly shaded sphere

```

glBegin(GL_TRIANGLES);
for (i = 0; i < 20; i++) {
    glNormal3fv(&vdata[tindices[i][0]][0]);
    glVertex3fv(&vdata[tindices[i][0]][0]);
    glNormal3fv(&vdata[tindices[i][1]][0]);
    glVertex3fv(&vdata[tindices[i][1]][0]);
    glNormal3fv(&vdata[tindices[i][2]][0]);
    glVertex3fv(&vdata[tindices[i][2]][0]);
}
glEnd();

```

Improving the Model

A twenty-sided approximation to a sphere doesn't look good unless the image of the sphere on the screen is quite small, but there's an easy way to increase the accuracy of the approximation. Imagine the icosahedron inscribed in a sphere, and subdivide the triangles as shown in Figure 2. The newly introduced vertices lie slightly inside the sphere, so push them to the surface by normalizing them (dividing them by a factor to make them have length 1). This subdivision process can be repeated for arbitrary accuracy. The three objects shown in Figure 2 use 20, 80, and 320 approximating triangles, respectively.

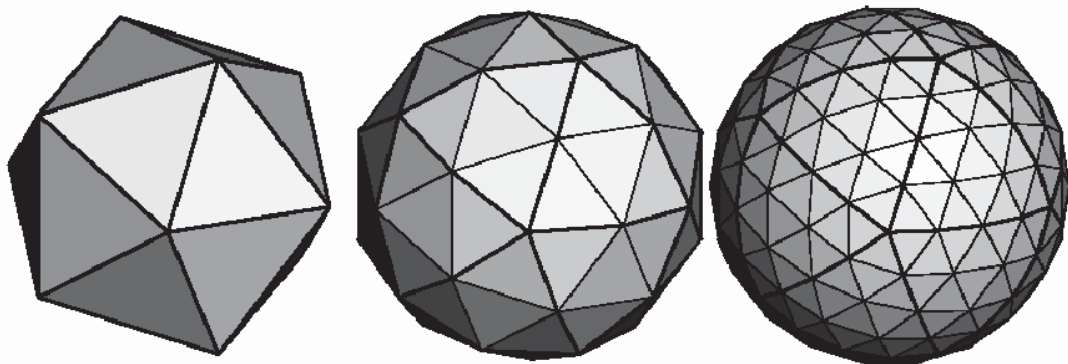


Figure 2: Subdividing to Improve a Polygonal Approximation to a Surface

Example 4 performs a single subdivision, creating an 80-sided spherical approximation.

Example 4 : Single Subdivision

```
void drawtriangle(float *v1, float *v2, float *v3)
{
    glBegin(GL_TRIANGLES);
        glVertex3fv(v1);
        glVertex3fv(v2);
        glVertex3fv(v3);
    glEnd();
}

void subdivide(float *v1, float *v2, float *v3)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;
    for (i = 0; i < 3; i++) {
        v12[i] = v1[i]+v2[i];
        v23[i] = v2[i]+v3[i];
        v31[i] = v3[i]+v1[i];
    }
    normalize(v12);
    normalize(v23);
    normalize(v31);
    drawtriangle(v1, v12, v31);
    drawtriangle(v2, v23, v12);
    drawtriangle(v3, v31, v23);
    drawtriangle(v12, v23, v31);
}

for (i = 0; i < 20; i++) {
    subdivide(&vdata[tindices[i][0]][0],
             &vdata[tindices[i][1]][0],
             &vdata[tindices[i][2]][0]);
}
```

Example 5 is a slight modification of Example 4 which recursively subdivides the triangles to the proper depth. If the depth value is 0, no subdivisions are performed, and the triangle is drawn as is. If the depth is 1, a single subdivision is performed, and so on.

Example 5 : Recursive Subdivision

```
void subdivide(float *v1, float *v2, float *v3, long depth)
{
    GLfloat v12[3], v23[3], v31[3];
    GLint i;
    if (depth == 0) {
        drawtriangle(v1, v2, v3);
        return;
    }
}
```

```

for (i = 0; i < 3; i++) {
    v12[i] = v1[i]+v2[i];
    v23[i] = v2[i]+v3[i];
    v31[i] = v3[i]+v1[i];
}
normalize(v12);
normalize(v23);
normalize(v31);
subdivide(v1, v12, v31, depth-1);
subdivide(v2, v23, v12, depth-1);
subdivide(v3, v31, v23, depth-1);
subdivide(v12, v23, v31, depth-1);
}

```

Generalized Subdivision

A recursive subdivision technique such as the one described in Example 5 can be used for other types of surfaces. Typically, the recursion ends either if a certain depth is reached or if some condition on the curvature is satisfied (highly curved parts of surfaces look better with more subdivision).

To look at a more general solution to the problem of subdivision, consider an arbitrary surface parameterized by two variables $u[0]$ and $u[1]$. Suppose that two routines are provided:

```

void surf(GLfloat u[2], GLfloat vertex[3], GLfloat normal[3]);
float curv(GLfloat u[2]);

```

If **surf()** is passed $u[]$, the corresponding three-dimensional vertex and normal vectors (of length 1) are returned. If $u[]$ is passed to **curv()**, the curvature of the surface at that point is calculated and returned. (See an introductory textbook on differential geometry for more information about measuring surface curvature.)

Example 6 shows the recursive routine that subdivides a triangle either until the maximum depth is reached or until the maximum curvature at the three vertices is less than some cutoff.

Example 6 : Generalized Subdivision

```

void subdivide(float u1[2], float u2[2], float u3[2], float cutoff, long depth)
{
    GLfloat v1[3], v2[3], v3[3], n1[3], n2[3], n3[3];
    GLfloat u12[2], u23[2], u32[2];
    GLint i;
    if (depth == maxdepth || (curv(u1) < cutoff && curv(u2) < cutoff && curv(u3) <
    cutoff)) {
        surf(u1, v1, n1); surf(u2, v2, n2); surf(u3, v3, n3);
        glBegin(GL_POLYGON);
            glNormal3fv(n1); glVertex3fv(v1);
            glNormal3fv(n2); glVertex3fv(v2);
            glNormal3fv(n3); glVertex3fv(v3);
        glEnd();
    }
}

```

```
        return;  
    }  
    for (i = 0; i < 2; i++) {  
        u12[i] = (u1[i] + u2[i])/2.0;  
        u23[i] = (u2[i] + u3[i])/2.0;  
        u31[i] = (u3[i] + u1[i])/2.0;  
    }  
    subdivide(u1, u12, u31, cutoff, depth+1);  
    subdivide(u2, u23, u12, cutoff, depth+1);  
    subdivide(u3, u31, u23, cutoff, depth+1);  
    subdivide(u12, u23, u31, cutoff, depth+1);  
}
```



Lecture No.40 Fractals

Fractal are geometric patterns that is repeated at ever smaller scales to produce irregular shapes and surfaces that can not be represented by classical geometry. Fractals are used in computer modeling of irregular patterns and structure in nature.

According to Webster's Dictionary a fractal is defined as being "derived from the Latin word *fractus* meaning broken, uneven: any of various extremely irregular curves or shapes that repeat themselves at any scale on which they are examined."

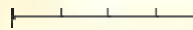
Mandelbrot, the discoverer of fractals gives two definitions:

- "I coined fractal from the Latin adjective *fractus*. The corresponding Latin verb *frangere* means 'to break:' to create irregular fragments. It is therefore sensible - and how appropriate for our needs! - that, in addition to 'fragmented' (as in fraction or refraction), *fractus* should also mean 'irregular,' both meanings being preserved in fragment"[3]
- Every set with a non-integer (*Hausdorff-Besicovitch*) dimension (D) is a fractal. However not every fractal has an integer D . A fractal is by definition a set for which D strictly exceeds the topological dimension (D^*).[3]

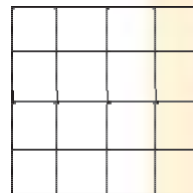
Hausdorff-Besicovitch(Fractal Dimension)

To understand the second definition we need to be able to understand the fractal dimension. So first we have to develop an understanding of "how to calculate the dimension of an object". Below we have three different objects.

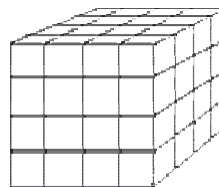
1. As you can see the line is broken into 4 smaller lines. Each of these lines is similar to the original line, but they are all $1/4$ the scale. This is the idea of *self similarity*.



2. The square below is also broken into smaller pieces. Each of which is $1/4$ th the size of the original. In this case it takes 16 of the smaller pieces to create the original.



3. As with the others the cube is also broken down into smaller cubes of $1/4$ the size of the original. It takes 64 of these smaller cubes to create the original cube.



By looking at this we begin to see a pattern:

$$4 = 4^1$$

$$16 = 4^2$$

$$64 = 4^3$$

This gives us the equation:

$$N = S^D$$

Where **N** is the number of small pieces that go into the larger one, **S** is the scale to which the smaller pieces compare to the larger one and **D** is the dimension.

We now have the tools to be able to calculate the dimension. Just solve for **D** in the previous equation. When we do this we find that the Dimension is:

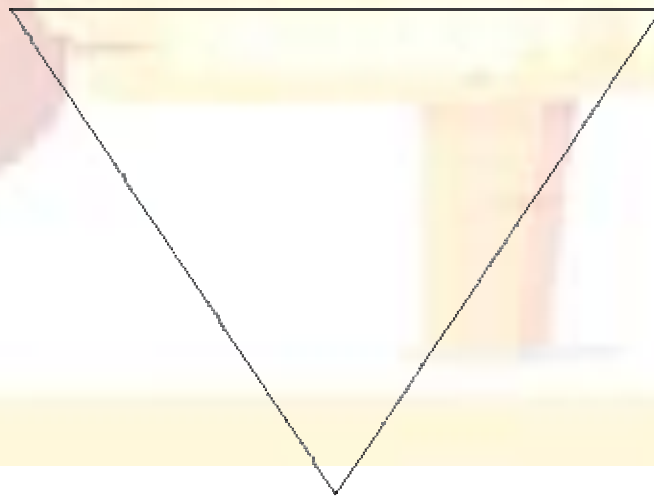
$$D = \log N / \log S$$

This dimension is the Hausdorff-Besicovitch dimension.

Koch Curve

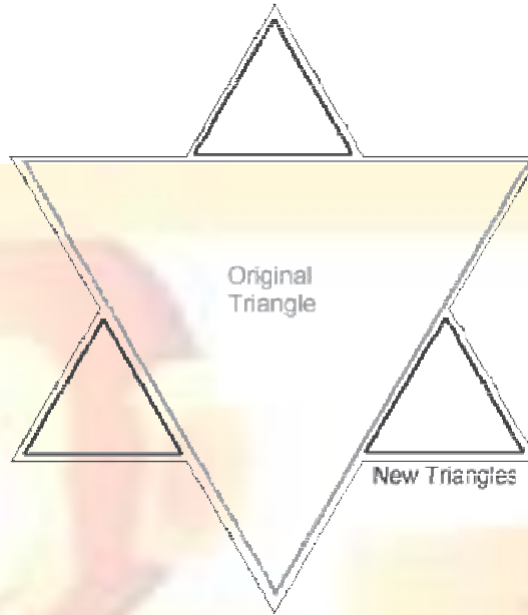
Euclidean Geometry is the stuff we all learn in school. It is the geometry of lines, planes, circles etc. It's simple and it works, and for a long time, mathematicians thought it was a reasonable representation of nature. However, people soon discovered that they could draw (or at least begin to draw) certain curves and surfaces that could not be described by the classical geometry.

How hard can it be to draw a curve? Let us attempt to describe. This is the Koch curve:



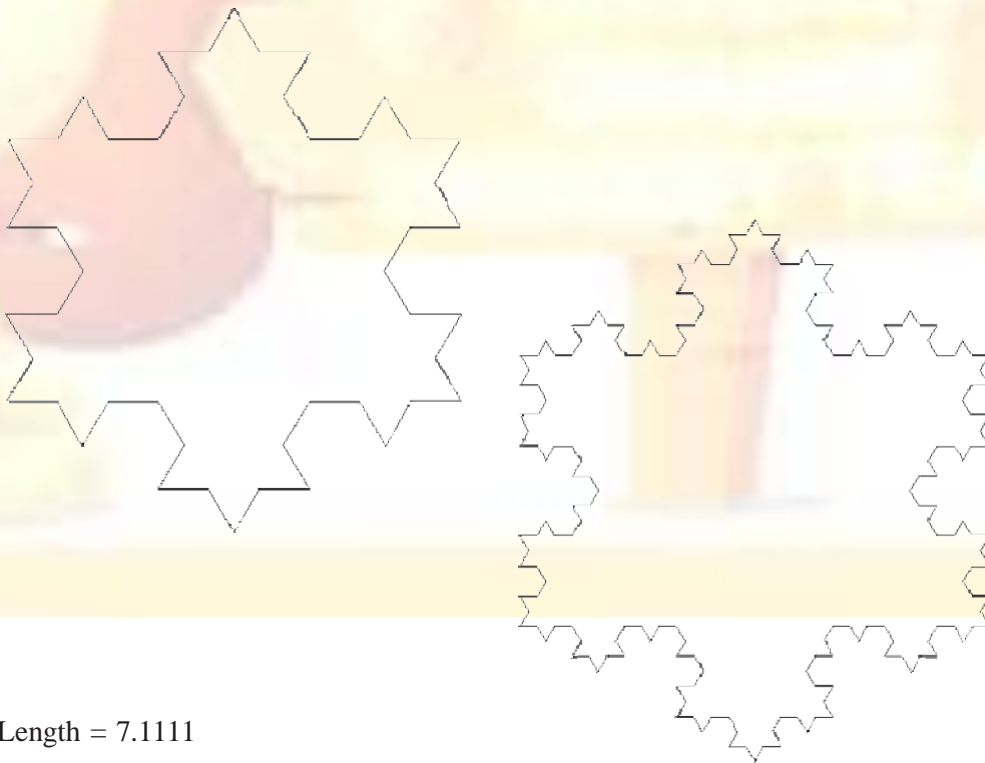
Draw a triangle.

If we say that each line is of length 1, then the total length of the curve is 3.

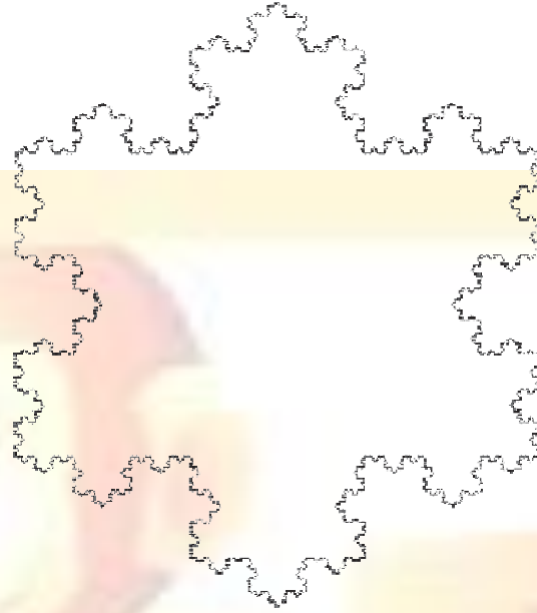


Now take each edge in turn and add another triangle, a third of the size. So now there are 12 edges and 12 points. The length of the curve is now 4. Repeat the process again, and again, forever.

Length = 5.3333



Length = 7.1111



Length = 12.6420

As we continue adding edges, the length of the curve increases. If we add edges forever, then length of the curve reaches infinity, but the whole curve nevertheless covers a finite area. The curve is infinitely detailed. No matter how closely we zoom into the image, it always shows up more detail.

Self Similarity

So what do these mathematical curiosities have to do with the real world? Well, everything as it turns out. Such objects turn up all the time in the natural world. Animals, plants, rocks, crystals and liquids all exhibit fractal properties and self similarity.

Let's take a look at a common plant, the fern. The fern is typical of many plants in that it exhibits self similarity. A fern consists of a leaf, which is made up from many similar, but smaller leaves, each of which, in turn, is made from even smaller leaves. The closer we look the more detail we see.

The following figure is a standard fern, which we may well find while being dragged on long walks in the country by your parents long before we are able to fully appreciate the beauty of nature. We will see the overall theme of repeating leaves. Each smaller leaf looks similar to the larger leaf.



Looking a little closer, we can see that those small leaves are made up from even smaller leaves.



Of course, in reality, a fern does have a smallest leaf, though we're sure every fern aspires to be like that one. What is interesting is that the program to generate this image is only a few lines long. The same tends to be true for all fractals. A very simple algorithm can explain an infinitely complex object.

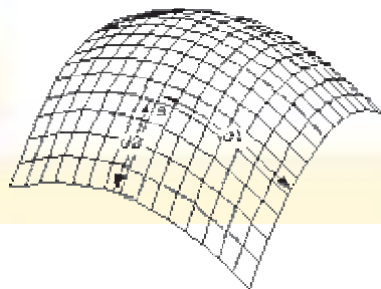
Fractal Geometry

Almost all geometric forms used for building man made objects belong to Euclidean geometry, they are comprised of lines, planes, rectangular volumes, arcs, cylinders, spheres, etc. These elements can be classified as belonging to an integer dimension, 1, 2, or 3. This concept of dimension can be described both intuitively and mathematically. Intuitively we say that a line is one dimensional because it only takes 1 number to uniquely define any point on it. That one number could be the distance from the start of the line. This applies equally well to the circumference of a circle, a curve, or the boundary of any object.



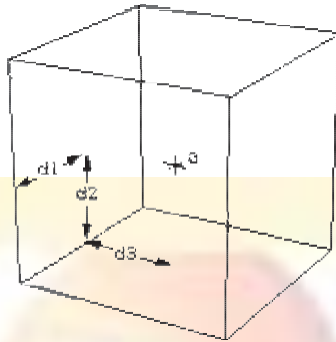
Any point "a" on a one dimensional curve can be represented by one number, the distance d from the start point.

A plane is two dimensional since in order to uniquely define any point on its surface we require two numbers. There are many ways to arrange the definition of these two numbers but we normally create an orthogonal coordinate system. Other examples of two dimensional objects are the surface of a sphere or an arbitrary twisted plane.



Any point "a" on a two dimensional surface can be uniquely represented by two numbers. One of the many possible methods is to grid the surface and measure two distances along the grid lines.

The volume of some solid object is 3 dimensional on the same basis as above, it takes three numbers to uniquely define any point within the object.



Any point "a" in three dimensions can be uniquely represented by three numbers. Typically these three numbers are the coordinates of the point using an orthogonal coordinate system.

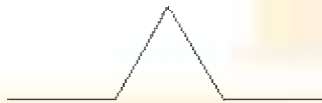
A more mathematical description of dimension is based on how the "size" of an object behaves as the linear dimension increases. In one dimension consider a line segment. If the linear dimension of the line segment is doubled then obviously the length (characteristic size) of the line has doubled. In two dimensions, if the linear dimensions of a rectangle for example is doubled then the characteristic size, the area, increases by a factor of 4. In three dimensions if the linear dimension of a box are doubled then the volume increases by a factor of 8. This relationship between dimension D , linear scaling L and the resulting increase in size S can be generalized and written as

$$S = L^D$$

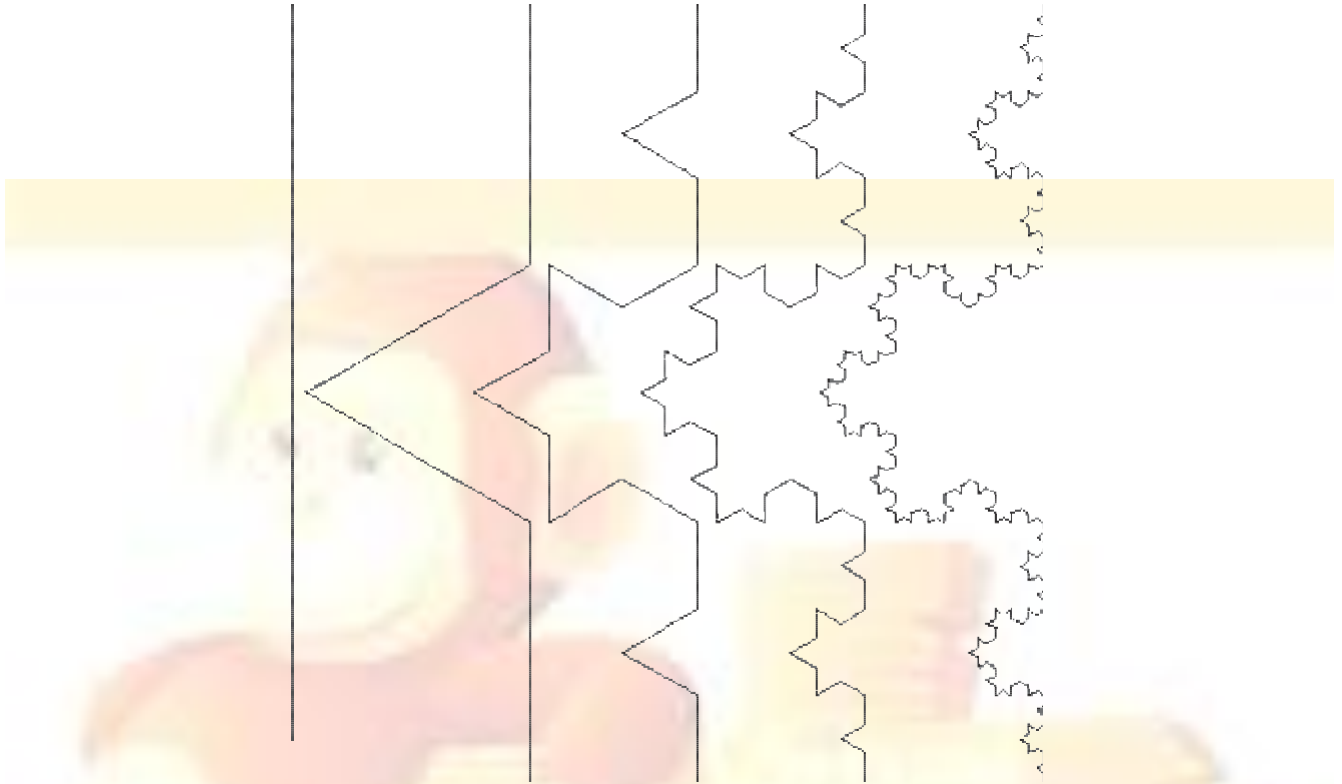
This is just telling us mathematically what we know from everyday experience. If we scale a two dimensional object for example then the area increases by the square of the scaling. If we scale a three dimensional object the volume increases by the cube of the scale factor. Rearranging the above gives an expression for dimension depending on how the size changes as a function of linear scaling, namely

$$D = \frac{\log(S)}{\log(L)}$$

In the examples above the value of D is an integer, 1, 2, or 3, depending on the dimension of the geometry. This relationship holds for all Euclidean shapes. There are however many shapes which do not conform to the integer based idea of dimension given above in both the intuitive and mathematical descriptions. That is, there are objects which appear to be curves for example but which a point on the curve cannot be uniquely described with just one number. If the earlier scaling formulation for dimension is applied the formula does not yield an integer. There are shapes that lie in a plane but if they are linearly scaled by a factor L , the area does not increase by L squared but by some non integer amount. These geometries are called fractals! One of the simpler fractal shapes is the von Koch snowflake. The method of creating this shape is to repeatedly replace each line segment with the following 4 line segments.



The process starts with a single line segment and continues for ever. The first few iterations of this procedure are shown below.



This demonstrates how a very simple generation rule for this shape can generate some unusual (fractal) properties. Unlike Euclidean shapes this object has detail at all levels. If one magnifies a Euclidean shape such as the circumference of a circle it becomes a different shape, namely a straight line. If we magnify this fractal more and more detail is uncovered, the detail is self similar or rather it is exactly self similar. Put another way, any magnified portion is identical to any other magnified portion.

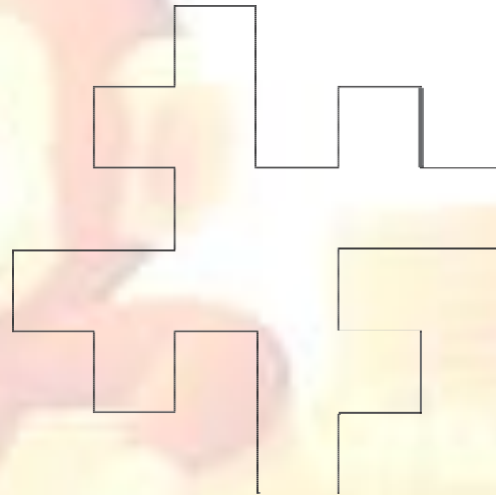
Note also that the "curve" on the right is not a fractal but only an approximation of one. This is no different from when one draws a circle, it is only an approximation to a perfect circle. At each iteration the length of the curve increases by a factor of $4/3$. Thus the limiting curve is of infinite length and indeed the length between any two points of the curve is infinite. This curve manages to compress an infinite length into a finite area of the plane without intersecting itself! Considering the intuitive notion of 1 dimensional shapes, although this object appears to be a curve with one starting point and one end point, it is not possible to uniquely specify any position along the curve with one number as we expect to be able to do with Euclidean curves which are 1 dimensional. Although the method of creating this curve is straightforward, there is no algebraic formula that describes the points on the curve. Some of the major differences between fractal and Euclidean geometry are outlined in the following table.

fractal	Euclidean
modern invention	traditional
no specific size or scale	based on a characteristic size or scale
appropriate for geometry in nature	suits description of man made objects
described by an algorithm	described by a usually simple formula

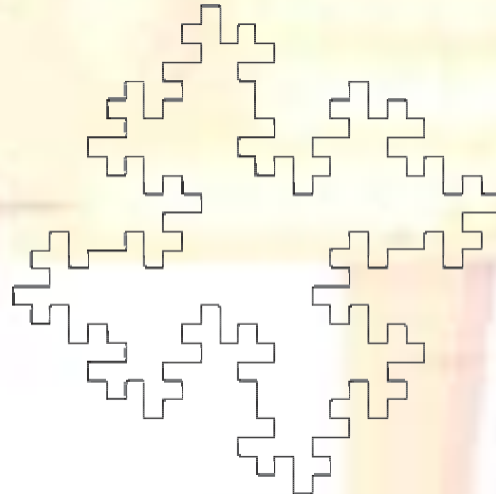
Firstly the recognition of fractal is very modern, they have only formally been studied in the last 10 years compared to Euclidean geometry which goes back over 2000 years.



The first iteration interpreted graphically is



The next iteration interpreted graphically is:



The following characters have a geometric interpretation.

Character	Meaning
F	Move forward by line length drawing a line
f	Move forward by line length without drawing a line
+	Turn left by turning angle
-	Turn right by turning angle
	Reverse direction (ie: turn by 180 degrees)
[Push current drawing state onto stack
]	Pop current drawing state from the stack
#	Increment the line width by line width increment
!	Decrement the line width by line width increment
@	Draw a dot with line width radius
{	Open a polygon
}	Close a polygon and fill it with fill colour
>	Multiply the line length by the line length scale factor
<	Divide the line length by the line length scale factor
&	Swap the meaning of + and -
(Decrement turning angle by turning angle increment
)	Increment turning angle by turning angle increment

Recent usage of L-Systems is for the creation of realistic looking objects that occur in nature and in particular the branching structure of plants. One of the important characteristics of L systems is that only a small amount of information is required to represent very complex objects. So while the bushes in figure 9 contain many thousands of lines they can be described in a database by only a few bytes of data, the actual bushes are only "grown" when required for visual presentation. Using suitably designed L-System algorithms it is possible to design the L-System production rules that will create a particular class of plant.

Further examples:

```

Axiom X
F --> FF
X --> F-
[(X)+X]+F|+FX] X
θ = 22.5

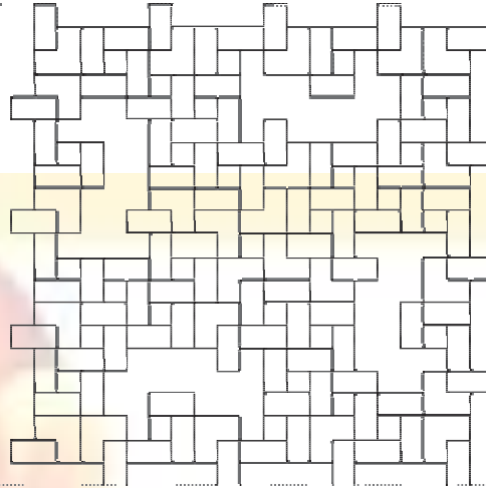
```



```

Axiom.F+F+F=F
F --> FF+~F+F+FF
θ = 90

```



```

Turning angle = 20
Axiom (depth=0) =
VZFFF
F --> F
V --> [+++W][---W|VV
W --> +X|-W|Z
X --> -W|+X|Z
Y --> VZ
Z --> [-FFF][+FFF]F

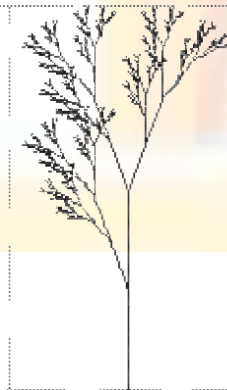
```



```

L String length = 491
Turning angle = 20
Axiom (depth=0) = X
F --> FF
X --> F[+X]F[-X]+X

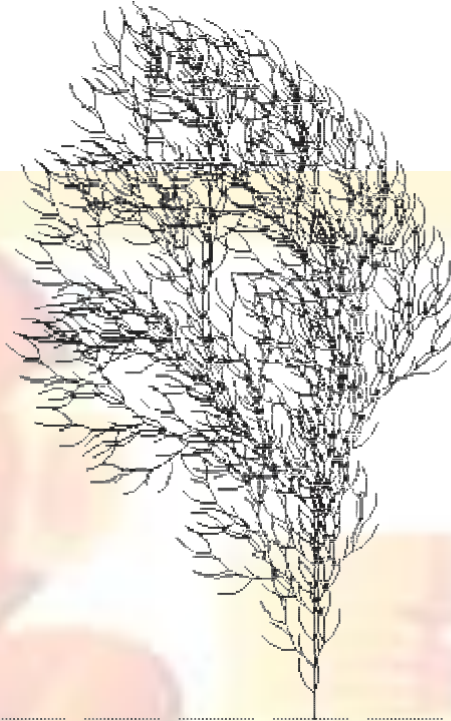
```



```

F --> FF:[+F-F-F]-|
F+F+F|
Turning angle = 22.5
Axiom (depth=0) = F

```



```

Y --> -FX-Y
Turning angle = 90
Axiom (depth=0) = FX
X --> X+YF+

```



```

Turning angle = 16
Axiom (depth=0) =
F1F1F1
0 < 0 > 0 --> 0
0 < 0 > 1 --> 1[-F1F1]
0 < 1 > 0 --> 1
0 < 1 > 1 --> 1
1 < 0 > 0 --> 0
1 < 0 > 1 --> 1F1
1 < 1 > 0 --> 1
1 < 1 > 1 --> F0
* < + > * --> -
* < - > * --> +

```

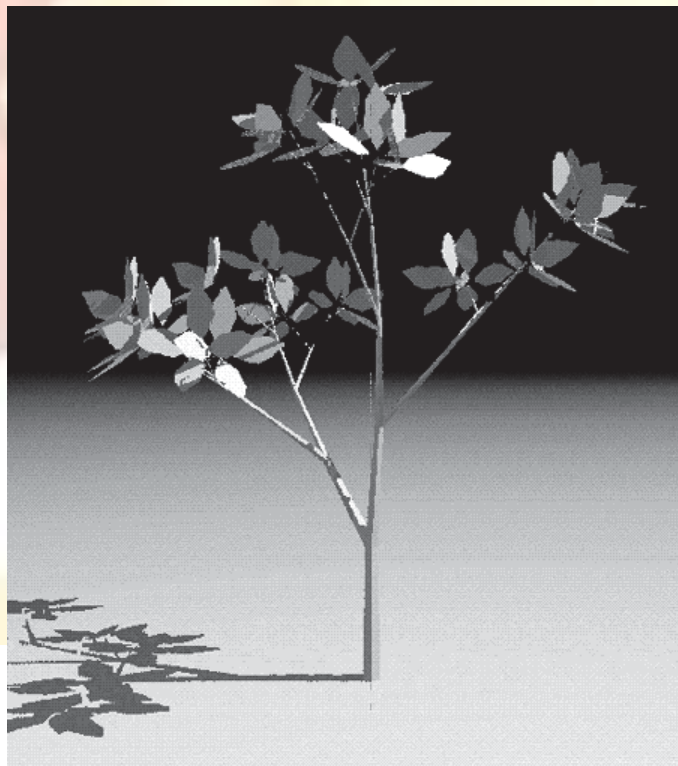


```

Turning angle = 30
Axiom (depth=0) = X
F --> FF P = 1
X --> F-([X]+X)+F(+FX)-X P = 0.5

```





Featured on the cover of the HPC (High Performance Computing) magazine, 3 August 2001.

IFS - Iterated Function Systems

Instead of working with lines as in L systems, IFS replaces polygons by other polygons as described by a generator. On every iteration each polygon is replaced by a suitably scaled, rotated, and translated version of the polygons in the generator. Figure 10 shows two such generators made of rectangles and the result after one and six iterations. From this geometric description it is also possible to derive a hop-along description which gives the image that would be created after iterating the geometric model to infinity. The description of this is a set of contractive transformations on a plane of the form

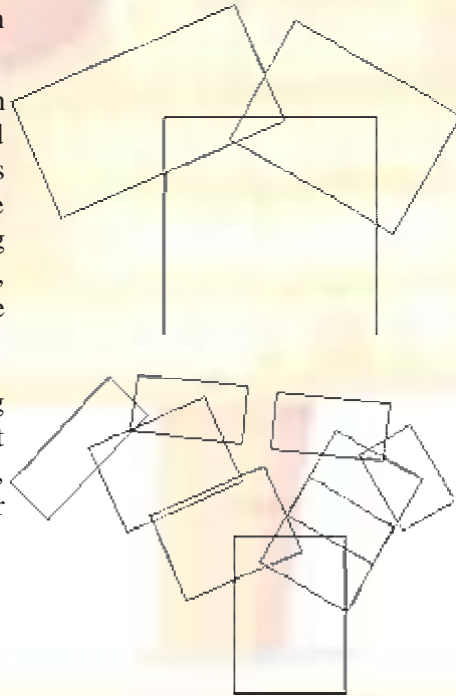
$$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x_{n-1} \\ y_{n-1} \end{pmatrix} + \begin{pmatrix} e \\ f \end{pmatrix}$$

each with an assigned probability. To run the system an initial point is chosen and on each iteration one of the transformation is chosen randomly according to the assigned probabilities, the resulting points (x_n, y_n) are drawn on the page. As in the case of L systems, if the IFS code for a desired image can be determined (by something called the Collage theorem) then large data compression ratios can be achieved. Instead of storing the geometry of the very complex object just the IFS generator needs to be stored and the image can be generated when required. The fundamental iterative process involves replacing rectangles with a series of rectangles called the generator. The rectangles are replaced by a suitably scaled, translated, and rotated version of the generator.

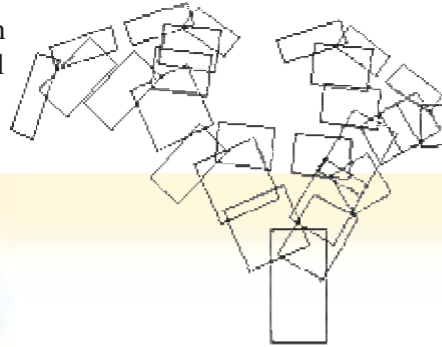
For example consider the generator on the right

It consists of three rectangles, each with its own center, dimensions, each and rotation angle. The initial conditions usually consist of a single square, the first iteration then consists of replacing this square by a suitably positioned, scaled and rotated version of the generator.

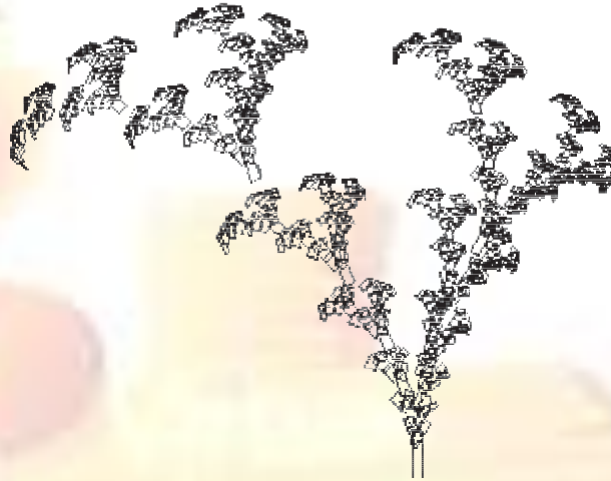
The next iteration involves replacing each of the rectangles in the current system by suitably positioned, scaled, and rotated versions of the generator resulting in the following



The next iteration replaces each rectangle above again by the initial generator as shown



and so on, three more iterations later



Hop-along or "The Chaos Game"

A technique exists by which the resulting form after an infinite number of iterations can be derived. This is a function of the form

$$\begin{aligned}x_{n+1} &\leftarrow f(x_n, y_n) \\ y_{n+1} &\leftarrow f(x_n, y_n)\end{aligned}$$

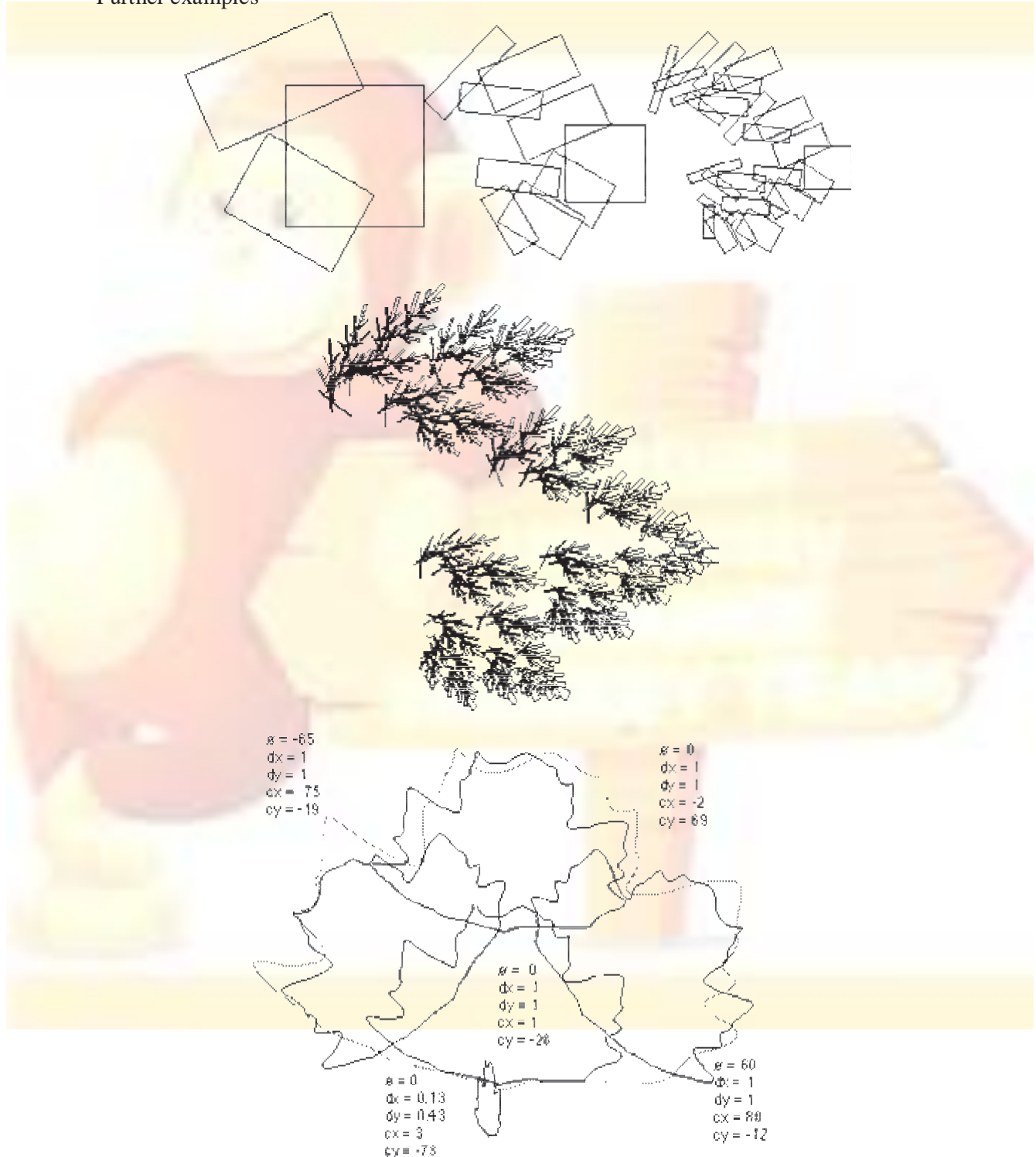
This gives a series of (x, y) points all which lie on the result of an infinite IFS. Although it still takes an infinite number of terms in this series to form the result the appearance can be readily appreciated after a modest number of terms (10000 say).

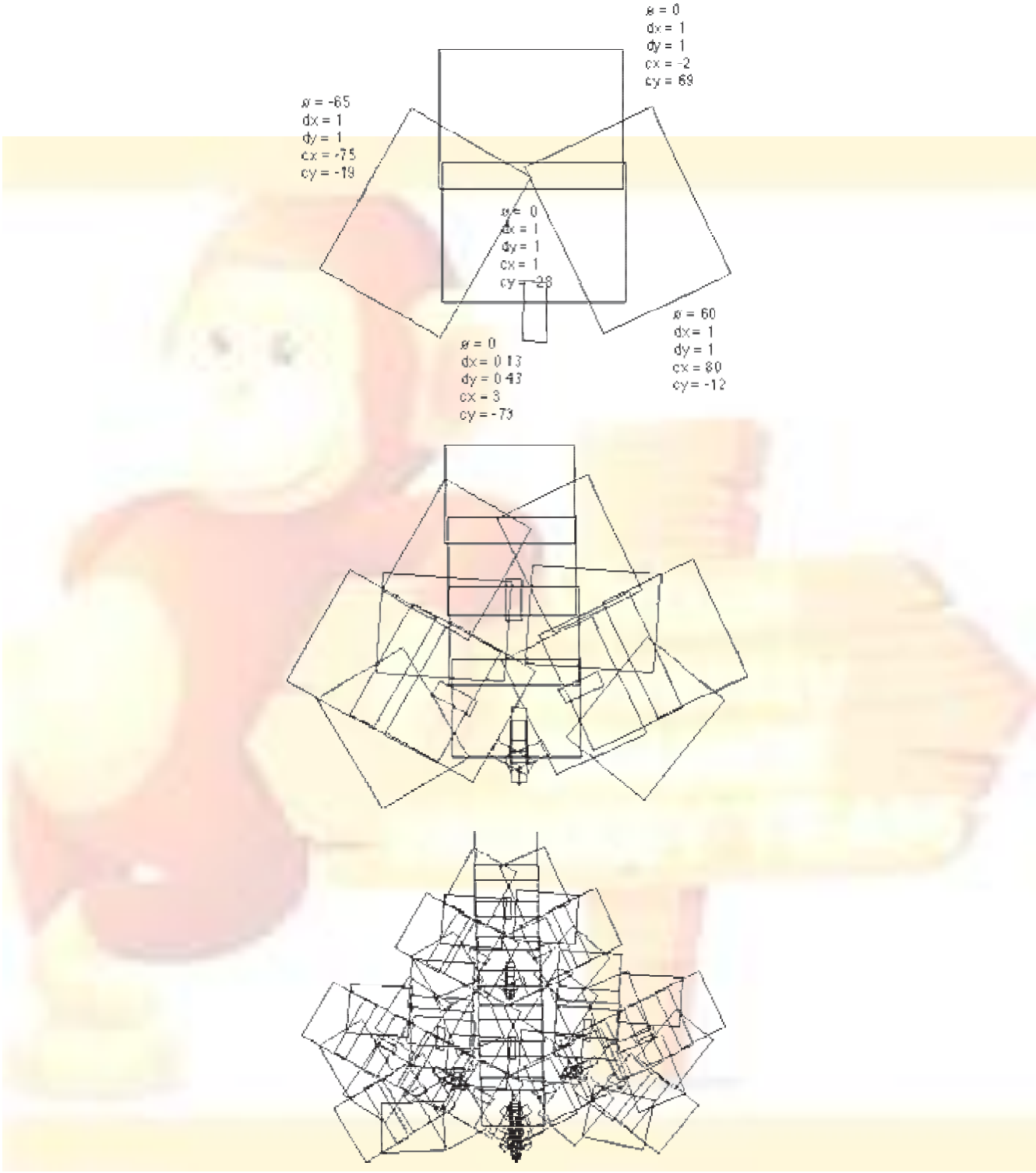


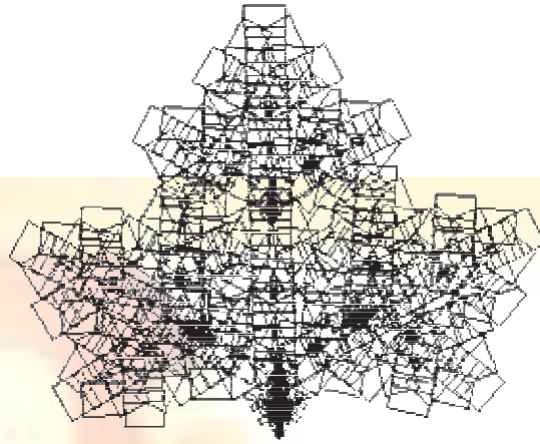
Note that with both methods it is possible to create the image at any scale. In many but not all cases zoomed in examples will exhibit self similarity at all scales. Applications generally involve data reduction for model files. If a generator can be found for a complex image then storing the generator and the rules of production results in a great deal of data reduction. For example the weed in the examples above might eventually

contain over 2000 rectangles but is completely specified by the characteristics of 3 rectangles, only 5 numbers, center (cx,cy), scale (sx,sy), and angle (ϕ) Note: it is not necessarily trivial to derive a rectangular generator for an arbitrary form, although it is possible to create a polygonal generator for any form.

Further examples



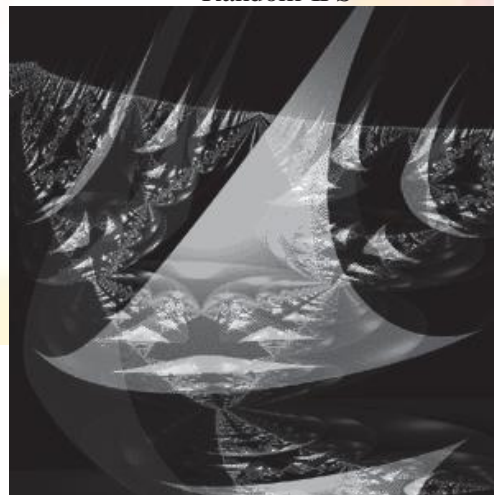




IFS Fern



Random IFS



Wada basins

Lecture No.41 Viewing

In this lecture we will practically implement viewing a *geometric model* in any orientation by transforming it in three-dimensional space and control the location in three-dimensional space from which the model is viewed and to Clip undesired portions of the model out of the scene that's to be viewed, manipulate the appropriate matrix stacks that control model transformation for viewing and project the model onto the screen, combine multiple transformations to mimic sophisticated systems in motion, such as a solar system or an articulated robot arm, Reverse or mimic the operations of the geometric processing pipeline 1 and we will also discuss on how to instruct OpenGL to draw the geometric models we want displayed in our scene. Now we must decide how we want to position the models in the scene, and we must choose a vantage point from which to view the scene. We can use the default positioning and vantage point, but most likely we want to specify them. Look at the image on the cover of this book. The program that produced that image contained a single geometric description of a building block. Each block was carefully positioned in the scene: Some blocks were scattered on the floor, some were stacked on top of each other on the table, and some were assembled to make the globe. Also, a particular viewpoint had to be chosen. Obviously, we wanted to look at the corner of the room containing the globe. But how far away from the scene - and where exactly - should the viewer be? We wanted to make sure that the final image of the scene contained a good view out the window, that a portion of the floor was visible, and that all the objects in the scene were not only visible but presented in an interesting arrangement. and how to use OpenGL to accomplish these tasks: how to position and orient models in three-dimensional space and how to establish the location - also in three-dimensional space - of the viewpoint. All of these factors help determine exactly what image appears on the screen. We want to remember that the point of computer graphics is to create a two-dimensional image of three-dimensional objects (it has to be two-dimensional because it's drawn on a flat screen), but we need to think in three-dimensional coordinates while making many of the decisions that determine what gets drawn on the screen. A common mistake people make when creating three-dimensional graphics is to start thinking too soon that the final image appears on a flat, two-dimensional screen. Avoid thinking about which pixels need to be drawn, and instead try to visualize three-dimensional space. Create your models in some three-dimensional universe that lies deep inside your computer, and let the computer do its job of calculating which pixels to color.

A series of three computer operations convert an object's three-dimensional coordinates to pixel positions on the screen. Transformations, which are represented by matrix multiplication, include modeling, viewing, and projection operations. Such operations include rotation, translation, scaling, reflecting, orthographic projection, and perspective projection. Generally, we use a combination of several transformations to draw a scene. Since the scene is rendered on a rectangular window, objects (or parts of objects) that lie outside the window must be clipped. In three-dimensional Computer graphics, clipping occurs by throwing out objects on one side of a clipping plane.

Finally, a correspondence must be established between the transformed coordinates and screen pixels. This is known as a *viewport* transformation.

Overview: The Camera Analogy

The transformation process to produce the desired scene for viewing is analogous to taking a photograph with a camera. As shown in Figure 1, the steps with a camera (or a computer) might be the following. Set up your tripod and pointing the camera at the scene (viewing transformation).

Arrange the scene to be photographed into the desired composition (modeling transformation).

Choose a camera lens or adjust the zoom (projection transformation).

Determine how large we want the final photograph to be - for example, we might want it enlarged (viewport transformation).

After these steps are performed, the picture can be snapped or the scene can be drawn.

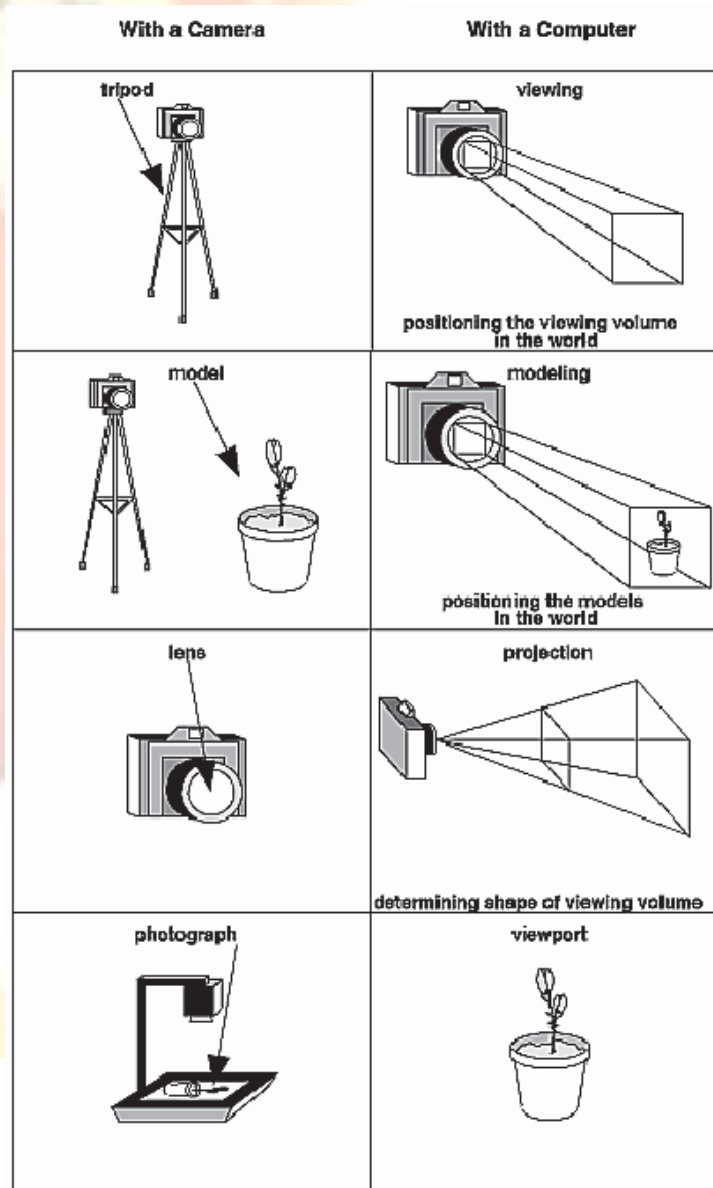


Figure 1: The Camera Analogy

Note that these steps correspond to the order in which we specify the desired transformations in our program, not necessarily the order in which the relevant mathematical operations are performed on an object's vertices. The viewing transformations must precede the modeling transformations in our code, but we can specify the projection and viewport transformations at any point before drawing occurs. Figure 2 shows the order in which these operations occur on our computer.

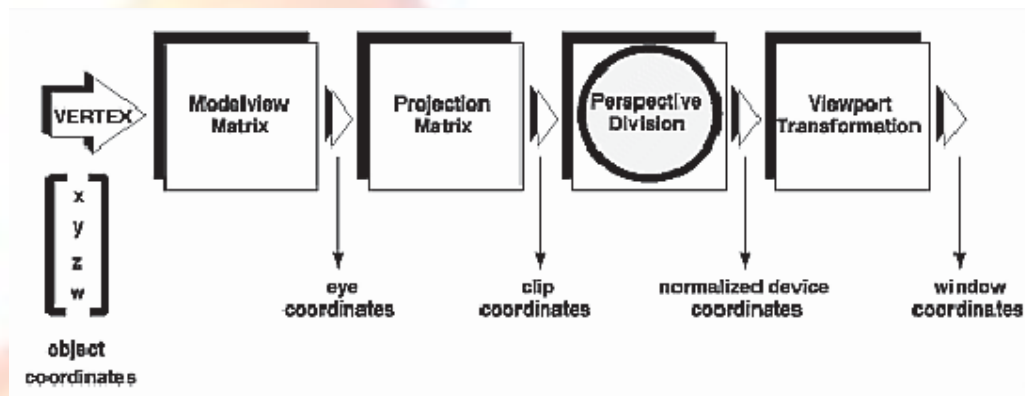


Figure 2: Stages of Vertex Transformation

To specify viewing, modeling, and projection transformations, we construct a 4×4 matrix M , which is then multiplied by the coordinates of each vertex v in the scene to accomplish the transformation

$$v' = Mv$$

(Remember that vertices always have four coordinates (x, y, z, w), though in most cases w is 1 and for two-dimensional data z is 0.) Note that viewing and modeling transformations are automatically applied to surface normal vectors, in addition to vertices. (Normal vectors are used only in eye coordinates.) This ensures that the normal vector's relationship to the vertex data is properly preserved.

The viewing and modeling transformations we specify are combined to form the modelview matrix, which is applied to the incoming object coordinates to yield eye coordinates. Next, if we've specified additional clipping planes to remove certain objects from the scene or to provide cutaway views of objects, these clipping planes are applied. After that, OpenGL applies the projection matrix to yield *clip coordinates*. This transformation defines a viewing volume; objects outside this volume are clipped so that they're not drawn in the final scene. After this point, the perspective division is performed by dividing coordinate values by w , to produce *normalized device coordinates*. Finally, the transformed coordinates are converted to window coordinates by applying the viewport transformation. We can manipulate the dimensions of the viewport to cause the final image to be enlarged, shrunk, or stretched. We might correctly suppose that the x and y coordinates are sufficient to determine which pixels need to be drawn on the screen. However, all the transformations are performed on the z coordinates as well. This way, at the end of this transformation process, the z values correctly reflect the depth of a given

vertex (measured in distance away from the screen). One use for this depth value is to eliminate unnecessary drawing. For example, suppose two vertices have the same x and y values but different z values. OpenGL can use this information to determine which surfaces are obscured by other surfaces and can then avoid drawing the hidden surfaces. As we've probably guessed by now, we need to know a few things about matrix mathematics to get the most out of this lecture as we have learnt from previous lectures.

A Simple Example: Drawing a Cube

Example 1 draws a cube that's scaled by a modeling transformation (see Figure 3). The viewing transformation, `gluLookAt()`, positions and aims the camera towards where the cube is drawn. A projection transformation and a viewport transformation are also specified. The rest of this section walks us through Example 1 and briefly explains the transformation commands it uses. The succeeding sections contain the complete, detailed discussion of all OpenGL's transformation commands.

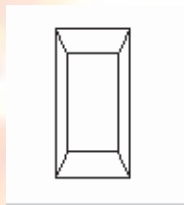


Figure 3: Transformed Cube

Example 1 : Transformed Cube

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_FLAT);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT);
    glColor3f (1.0, 1.0, 1.0);
    glLoadIdentity (); /* clear the matrix */
    /* viewing transformation */
    gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glScalef (1.0, 2.0, 1.0); /* modeling transformation */
    glutWireCube (1.0);
    glFlush ();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
```

```

glLoadIdentity ();
glFrustum (-1.0, 1.0, -1.0, 1.0, 1.5, 20.0);
glMatrixMode (GL_MODELVIEW);
}
int main(int argc, char** argv)
{
  glutInit(&argc, argv);
  glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
  glutInitWindowSize (500, 500);
  glutInitWindowPosition (100, 100);
  glutCreateWindow (argv[0]);
  init ();
  glutDisplayFunc(display);
  glutReshapeFunc(reshape);
  glutMainLoop();
  return 0;
}

```

The Viewing Transformation

Recall that the viewing transformation is analogous to positioning and aiming a camera. In this code example, before the viewing transformation can be specified, the current matrix is set to the identity matrix with **glLoadIdentity()**. This step is necessary since most of the transformation commands multiply the current matrix by the specified matrix and then set the result to be the current matrix. If we don't clear the current matrix by loading it with the identity matrix, we continue to combine previous transformation matrices with the new one we supply. In some cases, we do want to perform such combinations, but we also need to clear the matrix sometimes. In Example 1, after the matrix is initialized, the viewing transformation is specified with **gluLookAt()**. The arguments for this command indicate where the camera (or eye position) is placed, where it is aimed, and which way is up. The arguments used here place the camera at (0, 0, 5), aim the camera lens towards (0, 0, 0), and specify the up-vector as (0, 1, 0). The up-vector defines a unique orientation for the camera.

If **gluLookAt()** was not called, the camera has a default position and orientation. By default, the camera is situated at the origin, points down the negative z-axis, and has an up-vector of (0, 1, 0). So in Example 1, the overall effect is that **gluLookAt()** moves the camera 5 units along the z-axis.

The Modeling Transformation

We use the modeling transformation to position and orient the model. For example, we can rotate, translate, or scale the model - or perform some combination of these operations. In Example 1, **glScalef()** is the modeling transformation that is used. The arguments for this command specify how scaling should occur along the three axes. If all the arguments are 1.0, this command has no effect. In Example 1, the cube is drawn twice as large in the y direction. Thus, if one corner of the cube had originally been at (3.0, 3.0, 3.0), that corner would wind up being drawn at (3.0, 6.0, 3.0). The effect of this modeling transformation is to transform the cube so that it isn't a cube but a rectangular box.

Try This

Change the **gluLookAt()** call in Example 1 to the modeling transformation **glTranslatef()** with parameters (0.0, 0.0, -5.0). The result should look exactly the same as when we used **gluLookAt()**. Why are the effects of these two commands similar?

Note that instead of moving the camera (with a viewing transformation) so that the cube could be viewed, we could have moved the cube away from the camera (with a modeling transformation). This duality in the nature of viewing and modeling transformations is why we need to think about the effect of both types of transformations simultaneously. It doesn't make sense to try to separate the effects, but sometimes it's easier to think about them one way rather than the other. This is also why modeling and viewing transformations are combined into the *modelview matrix* before the transformations are applied.

Also note that the modeling and viewing transformations are included in the **display()** routine, along with the call that's used to draw the cube, **glutWireCube()**. This way, **display()** can be used repeatedly to draw the contents of the window if, for example, the window is moved or uncovered, and we've ensured that each time, the cube is drawn in the desired way, with the appropriate transformations. The potential repeated use of **display()** underscores the need to load the identity matrix before performing the viewing and modeling transformations, especially when other transformations might be performed between calls to **display()**.

The Projection Transformation

Specifying the projection transformation is like choosing a lens for a camera. We can think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it and to some extent how they look. This is equivalent to choosing among wide-angle, normal, and telephoto lenses, for example. With a wide-angle lens, we can include a wider scene in the final photograph than with a telephoto lens, but a telephoto lens allows us to photograph objects as though they're closer to us than they actually are. In computer graphics, we don't have to pay \$10,000 for a 2000-millimeter telephoto lens; once we've bought our graphics workstation, all we need to do is use a smaller number for our field of view. In addition to the field-of-view considerations, the projection transformation determines how objects are *projected* onto the screen, as its name suggests. Two basic types of projections are provided for us by OpenGL, along with several corresponding commands for describing the relevant parameters in different ways. One type is the *perspective* projection, which matches how we see things in daily life. Perspective makes objects that are farther away appear smaller; for example, it makes railroad tracks appear to converge in the distance. If we're trying to make realistic pictures, we'll want to choose perspective projection, which is specified with the **glFrustum()** command in this code example. The other type of projection is orthographic, which maps objects directly onto the screen without affecting their relative size. Orthographic projection is used in architectural and computer-aided design applications where the final image needs to reflect the measurements of objects rather than how they might look. Architects create perspective drawings to show how particular buildings or interior spaces look when viewed from various vantage points; the need for orthographic projection arises when blueprint plans or elevations are generated, which are used in the construction of buildings. Before **glFrustum()** can be called to set

the projection transformation, some preparation needs to happen. As shown in the **reshape()** routine in Example 1, the command called **glMatrixMode()** is used first, with the argument `GL_PROJECTION`. This indicates that the current matrix specifies the projection transformation; the following transformation calls then affect the projection matrix. As we can see, a few lines later **glMatrixMode()** is called again, this time with `GL_MODELVIEW` as the argument. This indicates that succeeding transformations now affect the modelview matrix instead of the projection matrix. Note that **glLoadIdentity()** is used to initialize the current projection matrix so that only the specified projection transformation has an effect. Now **glFrustum()** can be called, with arguments that define the parameters of the projection transformation. In this example, both the projection transformation and the viewport transformation are contained in the **reshape()** routine, which is called when the window is first created and whenever the window is moved or reshaped. This makes sense, since both projecting (the width to height aspect ratio of the projection viewing volume) and applying the viewport relate directly to the screen, and specifically to the size or aspect ratio of the window on the screen.

Try This

Change the **glFrustum()** call in Example 1 to the more commonly used Utility Library routine **gluPerspective()** with parameters (60.0, 1.0, 1.5, 20.0). Then experiment with different values, especially for fov(field of view), near and far plane.

Viewing and Modeling Transformations

Viewing and modeling transformations are inextricably related in OpenGL and are in fact combined into a single modelview matrix. (See "A Simple Example: Drawing a Cube.") One of the toughest problems newcomers to computer graphics face is understanding the effects of combined three-dimensional transformations. As we've already seen, there are alternative ways to think about transformations - do we want to move the camera in one direction, or move the object in the opposite direction? Each way of thinking about transformations has advantages and disadvantages, but in some cases one way more naturally matches the effect of the intended transformation. If we can find a natural approach for wer particular application, it's easier to visualize the necessary transformations and then write the corresponding code to specify the matrix manipulations. The first part of this section discusses how to think about transformations; later, specific commands are presented. For now, we use only the matrix-manipulation commands we've already seen. Finally, keep in mind that we must call **glMatrixMode()** with `GL_MODELVIEW` as its argument prior to performing modeling or viewing transformations.

Thinking about Transformations

Let's start with a simple case of two transformations: a 45-degree counterclockwise rotation about the origin around the z-axis, and a translation down the x-axis. Suppose that the object we're drawing is small compared to the translation (so that we can see the effect of the translation), and that it's originally located at the origin. If we rotate the object first and then translate it, the rotated object appears on the x-axis. If we translate it down the x-axis first, however, and then rotate about the origin, the object is on the line $y=x$, as shown in Figure 4. In general, the order of transformations is critical. If we do transformation A and then transformation B, we almost always get something different than if we do them in the opposite order.

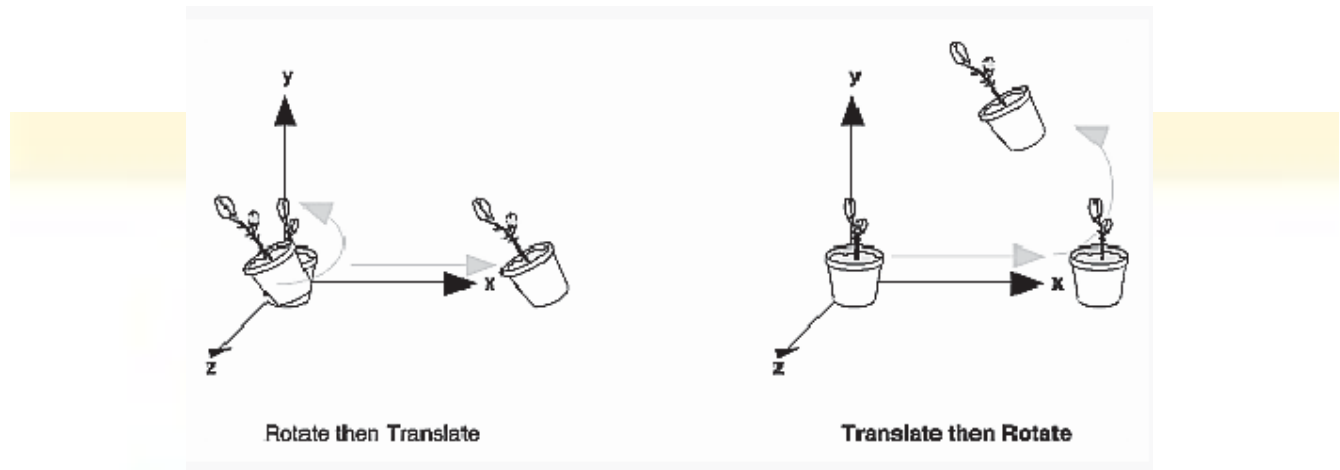


Figure 4 : Rotating First or Translating First

Now let's talk about the order in which we specify a series of transformations. All viewing and modeling transformations are represented as 4×4 matrices. Each successive `glMultMatrix*()` or transformation command multiplies a new 4×4 matrix M by the current modelview matrix C to yield CM . Finally, vertices v are multiplied by the current modelview matrix. This process means that the last transformation command called in our program is actually the first one applied to the vertices: CMv . Thus, one way of looking at it is to say that we have to specify the matrices in the reverse order. Like many other things, however, once we've gotten used to thinking about this correctly, backward will seem like forward. Consider the following code sequence, which draws a single point using three transformations:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glMultMatrixf(N); /* apply transformation N */
glMultMatrixf(M); /* apply transformation M */
glMultMatrixf(L); /* apply transformation L */
glBegin(GL_POINTS);
glVertex3f(v); /* draw transformed vertex v */
glEnd();
```

With this code, the modelview matrix successively contains I , N , NM , and finally NML , where I represents the identity matrix. The transformed vertex is $NMLv$. Thus, the vertex transformation is $N(M(Lv))$ - that is, v is multiplied first by L , the resulting Lv is multiplied by M , and the resulting MLv is multiplied by N . Notice that the transformations to vertex v effectively occur in the opposite order than they were specified. (Actually, only a single multiplication of a vertex by the modelview matrix occurs; in this example, the N , M , and L matrices are already multiplied into a single matrix before it's applied to v .)

Grand, Fixed Coordinate System

Thus, if we like to think in terms of a grand, fixed coordinate system - in which matrix multiplications affect the position, orientation, and scaling of our model - we have to think of the multiplications as occurring in the opposite order from how they appear in the code. Using the simple example shown on the left side of Figure 4 (a rotation about the origin and a translation along the x-axis), if we want the object to appear on the axis after the operations, the rotation must occur first, followed by the translation. To do this, we'll need to reverse the order of operations, so the code looks something like this (where \mathbf{R} is the rotation matrix and \mathbf{T} is the translation matrix):

```
glMatrixMode(GL_MODELVIEW);  
glLoadIdentity();  
glMultMatrixf(T); /* translation */  
glMultMatrixf(R); /* rotation */  
draw_the_object();
```

Moving a Local Coordinate System

Another way to view matrix multiplications is to forget about a grand, fixed coordinate system in which our model is transformed and instead imagine that a local coordinate system is tied to the object we're drawing. All operations occur relative to this changing coordinate system. With this approach, the matrix multiplications now appear in the natural order in the code. (Regardless of which analogy we're using, the code is the same, but how we think about it differs.) To see this in the translation-rotation example, begin by visualizing the object with a coordinate system tied to it. The translation operation moves the object and its coordinate system down the x-axis. Then, the rotation occurs about the (now-translated) origin, so the object rotates in place in its position on the axis. This approach is what we should use for applications such as articulated robot arms, where there are joints at the shoulder, elbow, and wrist, and on each of the fingers. To figure out where the tips of the fingers go relative to the body, we'd like to start at the shoulder, go down to the wrist, and so on, applying the appropriate rotations and translations at each joint. Thinking about it in reverse would be far more confusing. This second approach can be problematic, however, in cases where scaling occurs, and especially so when the scaling is non-uniform (scaling different amounts along the different axes). After uniform scaling, translations move a vertex by a multiple of what they did before, since the coordinate system is stretched. Non-uniform scaling mixed with rotations may make the axes of the local coordinate system non-perpendicular.

As mentioned earlier, we normally issue viewing transformation commands in our program before any modeling transformations. This way, a vertex in a model is first transformed into the desired orientation and then transformed by the viewing operation. Since the matrix multiplications must be specified in reverse order, the viewing commands need to come first. Note, however, that we don't need to specify either viewing or modeling transformations if we're satisfied with the default conditions. If there's no viewing transformation, the "camera" is left in the default position at the origin, pointed toward the negative z-axis; if there's no modeling transformation, the model isn't moved, and it retains its specified position, orientation, and size. Since the commands for performing modeling transformations can be used to perform viewing transformations,

modeling transformations are *discussed* first, even if viewing transformations are actually *issued* first. This order for discussion also matches the way many programmers think when planning their code: Often, they write all the code necessary to compose the scene, which involves transformations to position and orient objects correctly relative to each other. Next, they decide where they want the viewpoint to be relative to the scene they've composed, and then they write the viewing transformations accordingly.

Modeling Transformations

The three OpenGL routines for modeling transformations are **glTranslate*()**, **glRotate*()**, and **glScale*()**. As we might suspect, these routines transform an object (or coordinate system, if we're thinking of it that way) by moving, rotating, stretching, shrinking, or reflecting it. All three commands are equivalent to producing an appropriate translation, rotation, or scaling matrix, and then calling **glMultMatrix*()** with that matrix as the argument. However, these three routines might be faster than using **glMultMatrix*()**. OpenGL automatically computes the matrices for we. In the command summaries that follow, each matrix multiplication is described in terms of what it does to the vertices of a geometric object using the fixed coordinate system approach, and in terms of what it does to the local coordinate system that's attached to an object. **Translate**

```
void glTranslate{fd}(TYPE x, TYPE y, TYPE z);
```

Multiplies the current matrix by a matrix that moves (translates) an object by the given x, y, and z values (or moves the local coordinate system by the same amounts).

Figure 5 shows the effect of **glTranslate*()**.

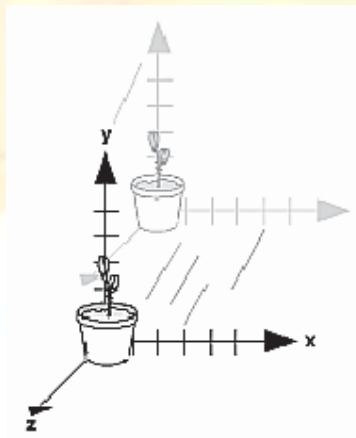


Figure 5 : Translating an Object

Note that using (0.0, 0.0, 0.0) as the argument for **glTranslate*()** is the identity operation - that is, it has no effect on an object or its local coordinate system.

Rotate

```
void glRotate{fd}(TYPE angle, TYPE x, TYPE y, TYPE z);
```

Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counterclockwise direction about the ray from the origin through the point (x, y, z). The angle parameter specifies the angle of rotation in degrees.

The effect of `glRotatef(45.0, 0.0, 0.0, 1.0)`, which is a rotation of 45 degrees about the z-axis, is shown in Figure 6.

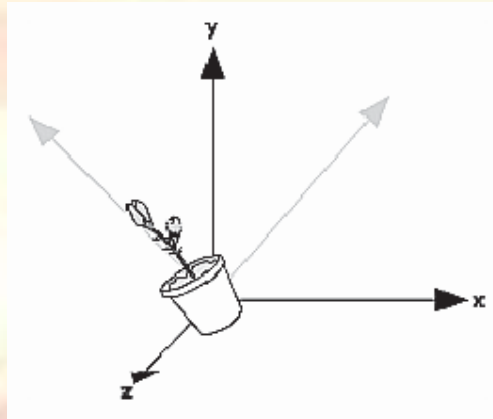


Figure 6 : Rotating an Object

Note that an object that lies farther from the axis of rotation is more dramatically rotated (has a larger orbit) than an object drawn near the axis. Also, if the angle argument is zero, the `glRotate*()` command has no effect.

Scale

```
void glScale(fd)(TYPEx, TYPE y, TYPEz);
```

Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each x, y, and z coordinate of every point in the object is multiplied by the corresponding argument x, y, or z. With the local coordinate system approach, the local coordinate axes are stretched, shrunk, or reflected by the x, y, and z factors, and the associated object is transformed with them.

Figure 7 shows the effect of `glScalef(2.0, -0.5, 1.0)`.

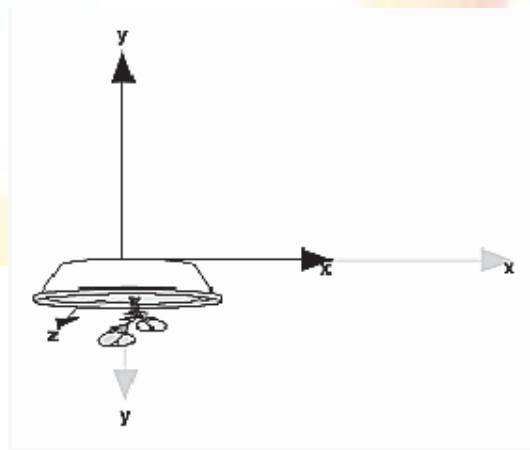


Figure 7 : Scaling and Reflecting an Object

glScale*() is the only one of the three modeling transformations that changes the apparent size of an object: Scaling with values greater than 1.0 stretches an object, and using values less than 1.0 shrinks it. Scaling with a -1.0 value reflects an object across an axis. The identity values for scaling are (1.0, 1.0, 1.0). In general, we should limit our use of **glScale*()** to those cases where it is necessary. Using **glScale*()** decreases the performance of lighting calculations, because the normal vectors have to be renormalized after transformation.

Note: A scale value of zero collapses all object coordinates along that axis to zero. It's usually not a good idea to do this, because such an operation cannot be undone. Mathematically speaking, the matrix cannot be inverted, and inverse matrices are required for certain lighting operations. Sometimes collapsing coordinates does make sense, however; the calculation of shadows on a planar surface is a typical application. In general, if a coordinate system is to be collapsed, the projection matrix should be used rather than the modelview matrix.

A Modeling Transformation Code Example

Example 2 is a portion of a program that renders a triangle four times, as shown in Figure 8. These are the four transformed triangles.

A solid wireframe triangle is drawn with no modeling transformation.

The same triangle is drawn again, but with a dashed line stipple and translated (to the left - along the negative x-axis).

A triangle is drawn with a long dashed line stipple, with its height (y-axis) halved and its width (x-axis) increased by 50%.

A rotated triangle, made of dotted lines, is drawn.

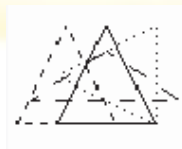


Figure 8 : Modeling Transformation Example

Example 2 : Using Modeling Transformations:

```
glLoadIdentity();
glColor3f(1.0, 1.0, 1.0);
draw_triangle(); /* solid lines */
glEnable(GL_LINE_STIPPLE); /* dashed lines */
glLineStipple(1, 0xF0F0);
glLoadIdentity();
glTranslatef(-20.0, 0.0, 0.0);
draw_triangle();
glLineStipple(1, 0xF00F); /*long dashed lines */
```

```
glLoadIdentity();
glScalef(1.5, 0.5, 1.0);
draw_triangle();
glLineStipple(1, 0x8888); /* dotted lines */
glLoadIdentity();
glRotatef(90.0, 0.0, 0.0, 1.0);
draw_triangle ();
glDisable (GL_LINE_STIPPLE);
```

Note the use of **glLoadIdentity()** to isolate the effects of modeling transformations; initializing the matrix values prevents successive transformations from having a cumulative effect. Even though using **glLoadIdentity()** repeatedly has the desired effect, it may be inefficient, because we may have to re specify viewing or modeling transformations.

Note: Sometimes, programmers who want a continuously rotating object attempt to achieve this by repeatedly applying a rotation matrix that has small values. The problem with this technique is that because of round-off errors, the product of thousands of tiny rotations gradually drifts away from the value we really want (it might even become something that isn't a rotation). Instead of using this technique, increment the angle and issue a new rotation command with the new angle at each update step.

Viewing Transformations

A viewing transformation changes the position and orientation of the viewpoint. If we recall the camera analogy, the viewing transformation positions the camera tripod, pointing the camera toward the model. Just as we move the camera to some position and rotate it until it points in the desired direction, viewing transformations are generally composed of translations and rotations. Also remember that to achieve a certain scene composition in the final image or photograph, we can either move the camera or move all the objects in the opposite direction. Thus, a modeling transformation that rotates an object counterclockwise is equivalent to a viewing transformation that rotates the camera clockwise, for example. Finally, keep in mind that the viewing transformation commands must be called before any modeling transformations are performed, so that the modeling transformations take effect on the objects first.

We can manufacture a viewing transformation in any of several ways, as described next. we can also choose to use the default location and orientation of the viewpoint, which is at the origin, looking down the negative z-axis.

Use one or more modeling transformation commands (that is, **glTranslate*()** and **glRotate*()**). We can think of the effect of these transformations as moving the camera position or as moving all the objects in the world, relative to a stationary camera.

Use the Utility Library routine **gluLookAt()** to define a line of sight. This routine encapsulates a series of rotation and translation commands.

Create our own utility routine that encapsulates rotations and translations. Some applications might require custom routines that allow we to specify the viewing transformation in a convenient way. For example, we might want to specify the roll,

pitch, and heading rotation angles of a plane in flight, or we might want to specify a transformation in terms of polar coordinates for a camera that's orbiting around an object.

Using `glTranslate*()` and `glRotate*()`

When we use modeling transformation commands to emulate viewing transformations, we're trying to move the viewpoint in a desired way while keeping the objects in the world stationary. Since the viewpoint is initially located at the origin and since objects are often most easily constructed there as well (see Figure 9), in general we have to perform some transformation so that the objects can be viewed. Note that, as shown in the figure, the camera initially points down the negative z-axis. (we're seeing the back of the camera.)

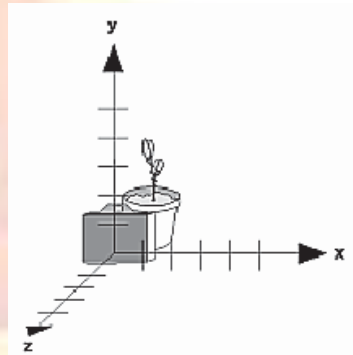


Figure 9 : Object and Viewpoint at the Origin

In the simplest case, we can move the viewpoint backward, away from the objects; this has the same effect as moving the objects forward, or away from the viewpoint. Remember that by default forward is down the negative z-axis; if we rotate the viewpoint, forward has a different meaning. So, to put 5 units of distance between the viewpoint and the objects by moving the viewpoint, as shown in Figure 10, use `glTranslatef(0.0, 0.0, -5.0)`;

This routine moves the objects in the scene -5 units along the z axis. This is also equivalent to moving the camera +5 units along the z axis.

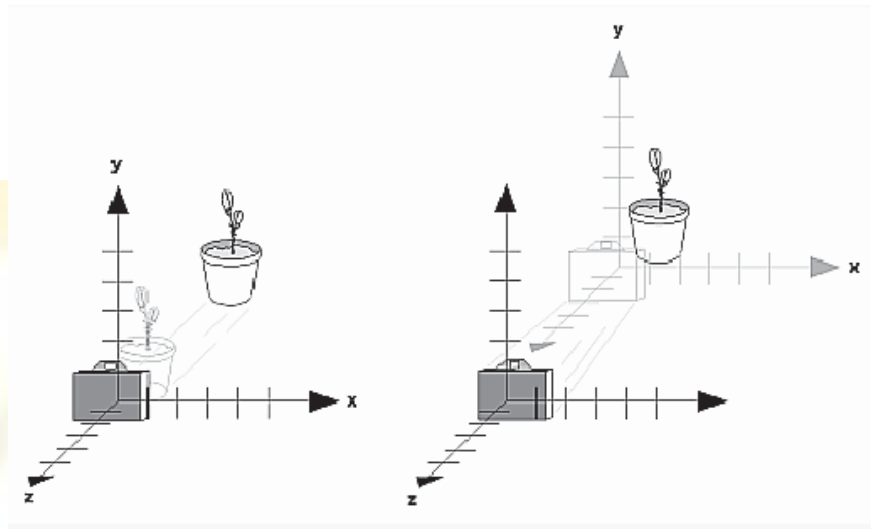


Figure 10 : Separating the Viewpoint and the Object

Now suppose we want to view the objects from the side. Should we issue a rotate command before or after the translate command? If we're thinking in terms of a grand, fixed coordinate system, first imagine both the object and the camera at the origin. We could rotate the object first and then move it away from the camera so that the desired side is visible. Since we know that with the fixed coordinate system approach, commands have to be issued in the opposite order in which they should take effect, we know that we need to write the translate command first in our code and follow it with the rotate command. Now let's use the local coordinate system approach. In this case, think about moving the object and its local coordinate system away from the origin; then, the rotate command is carried out using the now-translated coordinate system. With this approach, commands are issued in the order in which they're applied, so once again the translate command comes first. Thus, the sequence of transformation commands to produce the desired result is

```
glTranslatef(0.0, 0.0, -5.0);
glRotatef(90.0, 0.0, 1.0, 0.0);
```

If we're having trouble keeping track of the effect of successive matrix multiplications, try using both the fixed and local coordinate system approaches and see whether one makes more sense to us. Note that with the fixed coordinate system, rotations always occur about the grand origin, whereas with the local coordinate system, rotations occur about the origin of the local system. We might also try using the **gluLookAt()** utility routine described in the next section.

Using the **gluLookAt()** Utility Routine

Often, programmers construct a scene around the origin or some other convenient location, then they want to look at it from an arbitrary point to get a good view of it. As its name suggests, the **gluLookAt()** utility routine is designed for just this purpose. It takes three sets of arguments, which specify the location of the viewpoint, define a reference point toward which the camera is aimed, and indicate which direction is up.

Choose the viewpoint to yield the desired view of the scene. The reference point is typically somewhere in the middle of the scene. (If we've built our scene at the origin, the reference point is probably the origin.) It might be a little trickier to specify the correct up-vector. Again, if we've built some real-world scene at or around the origin and if we've been taking the positive y-axis to point upward, then that's our up-vector for **gluLookAt()**. However, if we're designing a flight simulator, up is the direction perpendicular to the plane's wings, from the plane toward the sky when the plane is right-side up on the ground.

The **gluLookAt()** routine is particularly useful when we want to pan across a landscape, for instance. With a viewing volume that's symmetric in both x and y, the (eyex, eyey, eyez) point specified is always in the center of the image on the screen, so we can use a series of commands to move this point slightly, thereby panning across the scene.

```
void gluLookAt(GLdouble eyex, GLdouble eyey, GLdouble eyez, GLdouble centerx,
GLdouble centery, GLdouble centerz, GLdouble upx, GLdouble upy, GLdouble upz);
```

Defines a viewing matrix and multiplies it to the right of the current matrix. The desired viewpoint is specified by eyex, eyey, and eyez. The centerx, centery, and centerz arguments specify any point along the desired line of sight, but typically they're some point in the center of the scene being looked at. The upx, upy, and upz arguments indicate which direction is up (that is, the direction from the bottom to the top of the viewing volume)

In the default position, the camera is at the origin, is looking down the negative z-axis, and has the positive y-axis as straight up. This is the same as calling

```
gluLookat (0.0, 0.0, 0.0, 0.0, 0.0, -100.0, 0.0, 1.0, 0.0);
```

The z value of the reference point is -100.0, but could be any negative z, because the line of sight will remain the same. In this case, we don't actually want to call **gluLookAt()**, because this is the default and we are already there! (The lines extending from the camera represent the viewing volume, which indicates its field of view.)

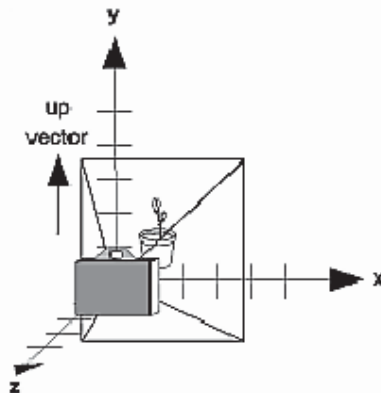


Figure 11 : Default Camera Position

Figure 12 shows the effect of a typical `gluLookAt()` routine. The camera position (`eyex`, `eyey`, `eyez`) is at (4, 2, 1). In this case, the camera is looking right at the model, so the reference point is at (2, 4, -3). An orientation vector of (2, 2, -1) is chosen to rotate the viewpoint to this 45-degree angle.

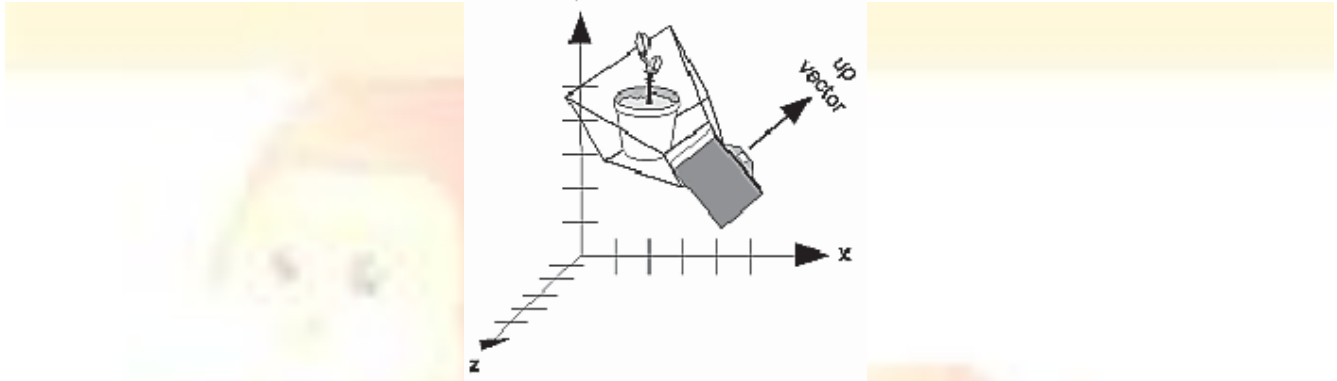


Figure 12 : Using `gluLookAt()`

So, to achieve this effect, call

```
gluLookAt(4.0, 2.0, 1.0, 2.0, 4.0, -3.0, 2.0, 2.0, -1.0);
```

Note that `gluLookAt()` is part of the Utility Library rather than the basic OpenGL library. This isn't because it's not useful, but because it encapsulates several basic OpenGL commands - specifically, `glTranslate*()` and `glRotate*()`. To see this, imagine a camera located at an arbitrary viewpoint and oriented according to a line of sight, both as specified with `gluLookAt()` and a scene located at the origin. To "undo" what `gluLookAt()` does, we need to transform the camera so that it sits at the origin and points down the negative z-axis, the default position. A simple translate moves the camera to the origin. We can easily imagine a series of rotations about each of the three axes of a fixed coordinate system that would orient the camera so that it pointed toward negative z values. Since OpenGL allows rotation about an arbitrary axis, we can accomplish any desired rotation of the camera with a single `glRotate*()` command.

Note: we can have only one active viewing transformation. we cannot try to combine the effects of two viewing transformations, any more than a camera can have two tripods. If we want to change the position of the camera, make sure we call `glLoadIdentity()` to wipe away the effects of any current viewing transformation.

Advanced

To transform any arbitrary vector so that it's coincident with another arbitrary vector (for instance, the negative z-axis), we need to do a little mathematics. The axis about which we want to rotate is given by the cross product of the two normalized vectors. To find the angle of rotation, normalize the initial two vectors. The cosine of the desired angle between the vectors is equal to the dot product of the normalized vectors. The angle of rotation around the axis given by the cross product is always between 0 and 180 degrees. Note that computing the angle between two normalized vectors by taking the inverse

cosine of their dot product is not very accurate, especially for small angles. But it should work well enough to get us started.

Creating a Custom Utility Routine

For some specialized applications, we might want to define our own transformation routine. Since this is rarely done and in any case is a fairly advanced topic, it's left mostly as an exercise for the reader. The following exercises suggest two custom viewing transformations that might be useful.

Try This

Suppose we're writing a flight simulator and we'd like to display the world from the point of view of the pilot of a plane. The world is described in a coordinate system with the origin on the runway and the plane at coordinates (x, y, z) . Suppose further that the plane has some roll, pitch, and heading (these are rotation angles of the plane relative to its center of gravity).

Show that the following routine could serve as the viewing transformation:

```
void pilotView(GLdouble planex, GLdouble planey,
GLdouble planez, GLdouble roll,
GLdouble pitch, GLdouble heading)
{
    glRotated(roll, 0.0, 0.0, 1.0);
    glRotated(pitch, 0.0, 1.0, 0.0);
    glRotated(heading, 1.0, 0.0, 0.0);
    glTranslated(-planex, -planey, -planez);
}
```

1 Suppose our application involves orbiting the camera around an object that's centered at the origin. In this case, we'd like to specify the viewing transformation by using polar coordinates. Let the distance variable define the radius of the orbit, or how far the camera is from the origin. (Initially, the camera is moved distance units along the positive z-axis.) The azimuth describes the angle of rotation of the camera about the object in the x-y plane, measured from the positive y-axis. Similarly, elevation is the angle of rotation of the camera in the y-z plane, measured from the positive z-axis. Finally, twist represents the rotation of the viewing volume around its line of sight. Show that the following routine could serve as the viewing transformation:

```
void polarView(GLdouble distance, GLdouble twist,
GLdouble elevation, GLdouble azimuth)
{
    glTranslated(0.0, 0.0, -distance);
    glRotated(-twist, 0.0, 0.0, 1.0);
    glRotated(-elevation, 1.0, 0.0, 0.0);
    glRotated(azimuth, 0.0, 0.0, 1.0);
}
```

Manipulating the Matrix Stacks

The modelview and projection matrices we've been creating, loading, and multiplying have only been the visible tips of their respective icebergs. Each of these matrices is actually the topmost member of a stack of matrices (see Figure 20).

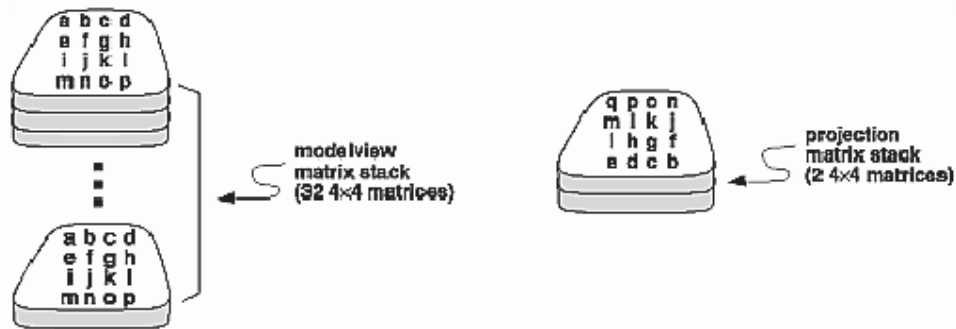


Figure 20 : Modelview and Projection Matrix Stacks

A stack of matrices is useful for constructing hierarchical models, in which complicated objects are constructed from simpler ones. For example, suppose we're drawing an automobile that has four wheels, each of which is attached to the car with five bolts. We have a single routine to draw a wheel and another to draw a bolt, since all the wheels and all the bolts look the same. These routines draw a wheel or a bolt in some convenient position and orientation, say centered at the origin with its axis coincident with the z axis. When we draw the car, including the wheels and bolts, we want to call the wheel-drawing routine four times with different transformations in effect each time to position the wheels correctly. As we draw each wheel, we want to draw the bolts five times, each time translated appropriately relative to the wheel. Suppose for a minute that all we have to do is draw the car body and the wheels. The English description of what we want to do might be something like this:

Draw the car body. Remember where we are, and translate to the right front wheel. Draw the wheel and throw away the last translation so our current position is back at the origin of the car body. Remember where we are, and translate to the left front wheel...

Similarly, for each wheel, we want to draw the wheel, remember where we are, and successively translate to each of the positions that bolts are drawn, throwing away the transformations after each bolt is drawn.

Since the transformations are stored as matrices, a matrix stack provides an ideal mechanism for doing this sort of successive remembering, translating, and throwing away. All the matrix operations that have been described so far (**glLoadMatrix()**, **glMultMatrix()**, **glLoadIdentity()** and the commands that create specific transformation matrices) deal with the current matrix, or the top matrix on the stack. We can control which matrix is on top with the commands that perform stack operations: **glPushMatrix()**, which copies the current matrix and adds the copy to the top of the

stack, and **glPopMatrix()**, which discards the top matrix on the stack, as shown in Figure 21. In effect, **glPushMatrix()** means "remember where we are" and **glPopMatrix()** means "go back to where we were."

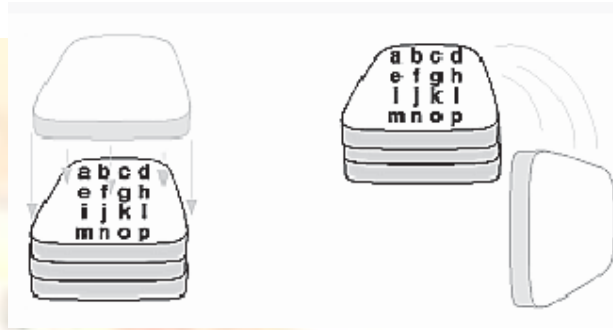


Figure 21 : Pushing and Popping the Matrix Stack

void glPushMatrix(void);

Pushes all matrices in the current stack down one level. The current stack is determined by glMatrixMode(). The topmost matrix is copied, so its contents are duplicated in both the top and second-from-the-top matrix. If too many matrices are pushed, an error is generated.

void glPopMatrix(void);

Pops the top matrix off the stack, destroying the contents of the popped matrix. What was the second-from-the-top matrix becomes the top matrix. The current stack is determined by glMatrixMode(). If the stack contains a single matrix, calling glPopMatrix() generates an error.

Example 4 draws an automobile, assuming the existence of routines that draw the car body, a wheel, and a bolt.

Example 4 : Pushing and Popping the Matrix

```
draw_wheel_and_bolts()
{
long i;
draw_wheel();
for(i=0;i<5;i++){
glPushMatrix();
glRotatef(72.0*i,0.0,0.0,1.0);
glTranslatef(3.0,0.0,0.0);
draw_bolt();
glPopMatrix();
}
}
draw_body_and_wheel_and_bolts()
{
draw_car_body();
glPushMatrix();
```

```
glTranslatef(40,0,30); /*move to first wheel position*/
draw_wheel_and_bolts();
glPopMatrix();
glPushMatrix();
glTranslatef(40,0,-30); /*move to 2nd wheel position*/
draw_wheel_and_bolts();
glPopMatrix();
... /*draw last two wheels similarly*/
}
```

This code assumes the wheel and bolt axes are coincident with the z-axis, that the bolts are evenly spaced every 72 degrees, 3 units (maybe inches) from the center of the wheel, and that the front wheels are 40 units in front of and 30 units to the right and left of the car's origin.

A stack is more efficient than an individual matrix, especially if the stack is implemented in hardware. When we push a matrix, we don't need to copy the current data back to the main process, and the hardware may be able to copy more than one element of the matrix at a time. Sometimes we might want to keep an identity matrix at the bottom of the stack so that we don't need to call **glLoadIdentity()** repeatedly.

Lecture No.42 Examples of Composing Several Transformations

This section demonstrates how to combine several transformations to achieve a particular result. The two examples discussed are a solar system, in which objects need to rotate on their axes as well as in orbit around each other, and a robot arm, which has several joints that effectively transform coordinate systems as they move relative to each other.

Building a Solar System

The program described in this section draws a simple solar system with a planet and a sun, both using the same sphere-drawing routine. To write this program, we need to use **glRotate*()** for the revolution of the planet around the sun and for the rotation of the planet around its own axis. We also need **glTranslate*()** to move the planet out to its orbit, away from the origin of the solar system. Remember that we can specify the desired size of the two spheres by supplying the appropriate arguments for the **glutWireSphere()** routine. To draw the solar system, we first want to set up a projection and a viewing transformation. For this example, **gluPerspective()** and **gluLookAt()** are used. Drawing the sun is straightforward, since it should be located at the origin of the grand, fixed coordinate system, which is where the sphere routine places it. Thus, drawing the sun doesn't require translation; we can use **glRotate*()** to make the sun rotate about an arbitrary axis. To draw a planet rotating around the sun, as shown in Figure 24, requires several modeling transformations. The planet needs to rotate about its own axis once a day. And once a year, the planet completes one revolution around the sun.

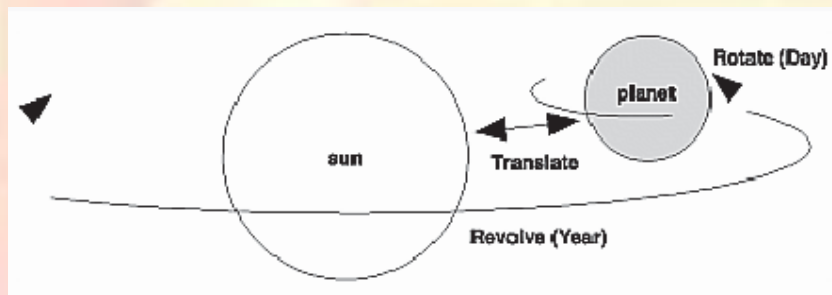


Figure 24 : Planet and Sun

To determine the order of modeling transformations, visualize what happens to the local coordinate system. An initial **glRotate*()** rotates the local coordinate system that initially coincides with the grand coordinate system. Next, **glTranslate*()** moves the local coordinate system to a position on the planet's orbit; the distance moved should equal the radius of the orbit. Thus, the initial **glRotate*()** actually determines where along the orbit the planet is (or what time of year it is). A second **glRotate*()** rotates the local coordinate system around the local axes, thus determining the time of day for the planet. Once we've issued all these transformation commands, the planet can be drawn.

In summary, these are the OpenGL commands to draw the sun and planet; the full program is shown in Example 6.

```
glPushMatrix();
glutWireSphere(1.0, 20, 16); /* draw sun */
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
```

```
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8); /* draw smaller planet */
glPopMatrix();
```

Example 6 : Planetary System:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
static int year = 0, day = 0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glColor3f (1.0, 1.0, 1.0);
glPushMatrix();
glutWireSphere(1.0, 20, 16); /* draw sun */
glRotatef ((GLfloat) year, 0.0, 1.0, 0.0);
glTranslatef (2.0, 0.0, 0.0);
glRotatef ((GLfloat) day, 0.0, 1.0, 0.0);
glutWireSphere(0.2, 10, 8); /* draw smaller planet */
glPopMatrix();
glutSwapBuffers();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective(60.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt (0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0);
}
void keyboard (unsigned char key, int x, int y)
{
switch (key) {
case `d`:
day = (day + 10) % 360;
glutPostRedisplay();
break;
case `D`:
day = (day - 10) % 360;
glutPostRedisplay();
break;
case `y`:
year = (year + 5) % 360;
```

```

glutPostRedisplay();
break;
case `Y':
year = (year - 5) % 360;
glutPostRedisplay();
break;
default:
break;
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutKeyboardFunc(keyboard);
glutMainLoop();
return 0;
}

```

Try This

Try adding a moon to the planet. Or try several moons and additional planets. Hint: Use **glPushMatrix()** and **glPopMatrix()** to save and restore the position and orientation of the coordinate system at appropriate moments. If we're going to draw several moons around a planet, we need to save the coordinate system prior to positioning each moon and restore the coordinate system after each moon is drawn.

Try tilting the planet's axis.

Building an Articulated Robot Arm

This section discusses a program that creates an articulated robot arm with two or more segments. The arm should be connected with pivot points at the shoulder, elbow, or other joints. Figure 25 shows a single joint of such an arm.

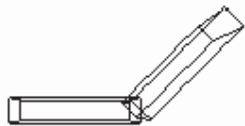


Figure 25 : Robot Arm

We can use a scaled cube as a segment of the robot arm, but first we must call the appropriate modeling transformations to orient each segment. Since the origin of the local coordinate system is initially at the center of the cube, we need to move the local

coordinate system to one edge of the cube. Otherwise, the cube rotates about its center rather than the pivot point.

After we call **glTranslate*()** to establish the pivot point and **glRotate*()** to pivot the cube, translate back to the center of the cube. Then the cube is scaled (flattened and widened) before it is drawn. The **glPushMatrix()** and **glPopMatrix()** restrict the effect of **glScale*()**. Here's what our code might look like for this first segment of the arm (the entire program is shown in Example 7):

```
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

To build a second segment, we need to move the local coordinate system to the next pivot point. Since the coordinate system has previously been rotated, the x-axis is already oriented along the length of the rotated arm. Therefore, translating along the x-axis moves the local coordinate system to the next pivot point. Once it's at that pivot point, we can use the same code to draw the second segment as we used for the first one. This can be continued for an indefinite number of segments (shoulder, elbow, wrist, fingers).

```
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
```

Example 7 : Robot Arm:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include <GL/glut.h>
static int shoulder = 0, elbow = 0;
void init(void)
{
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_FLAT);
}
void display(void)
{
glClear (GL_COLOR_BUFFER_BIT);
glPushMatrix();
glTranslatef (-1.0, 0.0, 0.0);
glRotatef ((GLfloat) shoulder, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
```

```

glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
glTranslatef (1.0, 0.0, 0.0);
glRotatef ((GLfloat) elbow, 0.0, 0.0, 1.0);
glTranslatef (1.0, 0.0, 0.0);
glPushMatrix();
glScalef (2.0, 0.4, 1.0);
glutWireCube (1.0);
glPopMatrix();
glPopMatrix();
glSwapBuffers();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity ();
gluPerspective(65.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
glTranslatef (0.0, 0.0, -5.0);
}
void keyboard (unsigned char key, int x, int y)
{
switch (key) {
case `s`: /* s key rotates at shoulder */
shoulder = (shoulder + 5) % 360;
glutPostRedisplay();
break;
case `S`:
shoulder = (shoulder - 5) % 360;
glutPostRedisplay();
break;
case `e`: /* e key rotates at elbow */
elbow = (elbow + 5) % 360;
glutPostRedisplay();
break;
case `E`:
elbow = (elbow - 5) % 360;
glutPostRedisplay();
break;
default:
break;
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_DOUBLE | GLUT_RGB);

```

```
glutInitWindowSize (500, 500);  
glutInitWindowPosition (100, 100);  
glutCreateWindow (argv[0]);  
init ();  
glutDisplayFunc(display);  
glutReshapeFunc(reshape);  
glutKeyboardFunc(keyboard);  
glutMainLoop();  
return 0;  
}
```

Try This

Modify Example 7 to add additional segments onto the robot arm. 1

Modify Example 7 to add additional segments at the same position. For example, give the robot arm several "fingers" at the wrist, as shown in Figure 26. Hint: Use **glPushMatrix()** and **glPopMatrix()** to save and restore the position and orientation of the coordinate system at the wrist. If we're going to draw fingers at the wrist, we need to save the current matrix prior to positioning each finger and restore the current matrix after each finger is drawn.

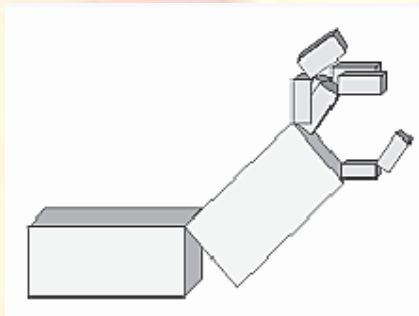


Figure 26: Robot Arm with Fingers

Lecture No.43 Real-World and OpenGL Lighting

When we look at a physical surface, our eye's perception of the color depends on the distribution of photon energies that arrive and trigger our cone cells. Those photons come from a light source or combination of sources, some of which are absorbed and some are reflected by the surface. In addition, different surfaces may have very different properties - some are shiny and preferentially reflect light in certain directions, while others scatter incoming light equally in all directions. Most surfaces are somewhere in between.

OpenGL approximates light and lighting as if light can be broken into red, green, and blue components. Thus, the color of light sources is characterized by the amount of red, green, and blue light they emit, and the material of surfaces is characterized by the percentage of the incoming red, green, and blue components that is reflected in various directions. The OpenGL lighting equations are just an approximation but one that works fairly well and can be computed relatively quickly. If we desire a more accurate (or just different) lighting model, we have to do our own calculations in software. Such software can be enormously complex, as a few hours of reading any optics textbook should convince us. In the OpenGL lighting model, the light in a scene comes from several light sources that can be individually turned on and off. Some light comes from a particular direction or position, and some light is generally scattered about the scene. For example, when we turn on a light bulb in a room, most of the light comes from the bulb, but some light comes after bouncing off one, two, three, or more walls. This bounced light (called ambient) is assumed to be so scattered that there is no way to tell its original direction, but it disappears if a particular light source is turned off.

Finally, there might be a general ambient light in the scene that comes from no particular source, as if it had been scattered so many times that its original source is impossible to determine. In the OpenGL model, the light sources have an effect only when there are surfaces that absorb and reflect light. Each surface is assumed to be composed of a material with various properties. A material might emit its own light (like headlights on an automobile), it might scatter some incoming light in all directions, and it might reflect some portion of the incoming light in a preferential direction like a mirror or other shiny surface. The OpenGL lighting model considers the lighting to be divided into four independent components: emissive, ambient, diffuse and specular. All four components are computed independently and then added together.

A Simple Example: Rendering a Lit Sphere

These are the steps required to add lighting to our scene. Define **NORMAL** vectors for each vertex of all the objects. These **NORMALS** determine the orientation of the object relative to the light sources.

Create, select, and position one or more light sources.

Create and select a *lighting model*, which defines the level of global ambient light and the effective location of the viewpoint (for the purposes of lighting calculations)

Define material properties for the objects in the scene.

Example 1 accomplishes these tasks. It displays a sphere illuminated by a single light source, as shown earlier in Figure 1.

Example 1 : Drawing a Lit Sphere:

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

void init(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat mat_shininess[] = { 50.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glShadeModel (GL_SMOOTH);
    glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
    glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    glEnable(GL_DEPTH_TEST);
}

void display(void)
{
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glutSolidSphere (1.0, 20, 16);
    glFlush ();
}

void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    if (w <= h)
        glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w, 1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
    else
        glOrtho (-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMainLoop();
    return 0;
}
```

The lighting-related calls are in the **init()** command; they're discussed briefly in the following paragraphs and in more detail later in the chapter. One thing to note about Example 1 is that it uses RGBA color mode, not color-index mode. The OpenGL lighting calculation is different for the two modes, and in fact the lighting capabilities are more limited in color-index mode. Thus, RGBA is the preferred mode when doing lighting.

Define Normal Vectors for Each Vertex of Every Object

An object's **NORMALS** determine its orientation relative to the light sources. For each vertex, OpenGL uses the assigned normal to determine how much light that particular vertex receives from each light source. In this example, the **NORMALS** for the sphere are defined as part of the **glutSolidSphere()** routine. (recall "Normal Vectors")

Create, Position, and Enable One or More Light Sources

Example 1 uses only one, white light source; its location is specified by the **glLightfv()** call. This example uses the default color for light zero (**GL_LIGHT0**), which is white; if we want a differently colored light, use **glLight*()** to indicate this. We can include at least eight different light sources in our scene of various colors; the default color of these other lights is black. (The particular implementation of OpenGL we're using might allow more than eight.) we can also locate the lights wherever we desire - we can position them near the scene, as a desk lamp would be, or an infinite distance away, like the sun. In addition, we can control whether a light produces a narrow, focused beam or a wider beam. Remember that each light source adds significantly to the calculations needed to render the scene, so performance is affected by the number of lights in the scene.

After we've defined the characteristics of the lights we want, we have to turn them on with the **glEnable()** command. We also need to call **glEnable()** with **GL_LIGHTING** as a parameter to prepare OpenGL to perform lighting calculations.

Select a Lighting Model

As we might expect, the **glLightModel*()** command describes the parameters of a lighting model. In Example 1, the only element of the lighting model that's defined explicitly is the global ambient light. The lighting model also defines whether the viewer of the scene should be considered to be an infinite distance away or local to the scene, and whether lighting calculations should be performed differently for the front and back surfaces of objects in the scene. Example 1 uses the default settings for these two aspects of the model - an infinite viewer and one-sided lighting. Using a local viewer adds significantly to the complexity of the calculations that must be performed, because OpenGL must calculate the angle between the viewpoint and each object. With an infinite viewer, however, the angle is ignored, and the results are slightly less realistic. Further, since in this example, the back surface of the sphere is never seen (it's the inside of the sphere), one-sided lighting is sufficient.

Define Material Properties for the Objects in the Scene

An object's material properties determine how it reflects light and therefore what material it seems to be made of. Because the interaction between an object's material surface and incident light is complex, specifying material properties so that an object has a certain desired appearance is an art. We can specify a material's ambient, diffuse, and specular colors and how shiny it is. In this example, only these last two material properties - the specular material color and shininess - are explicitly specified (with the **glMaterialfv()** calls).

Some Important Notes

As we write our own lighting program, remember that we can use the default values for some lighting parameters; others need to be changed. Also, don't forget to enable whatever lights we define and to enable lighting calculations. Finally, remember that we might be able to use display lists to maximize efficiency as we change lighting conditions.

Creating Light Sources

Light sources have a number of properties, such as color, position, and direction. The following sections explain how to control these properties and what the resulting light looks like. The command used to specify all properties of lights is **glLight*()**; it takes three arguments: to identify the light whose property is being specified, the property, and the desired value for that property.

```
void glLight{if}(GLenum light, GLenum pname, TYPEparam);
void glLight{if}v(GLenum light, GLenum pname, TYPE *param);
```

Creates the light specified by light, which can be GL_LIGHT0, GL_LIGHT1, ... , or GL_LIGHT7. The characteristic of the light being set is defined by pname, which specifies a named parameter (see Table 1). param indicates the values to which the pname characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued light characteristics.

Table 1 : Default Values for pname Parameter of glLight*()

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.0, 0.0, 0.0, 1.0)	ambient RGBA intensity of light
GL_DIFFUSE	(1.0, 1.0, 1.0, 1.0)	diffuse RGBA intensity of light
GL_SPECULAR	(1.0, 1.0, 1.0, 1.0)	specular RGBA intensity of light
GL_POSITION	(0.0, 0.0, 1.0, 0.0)	(x, y, z, w) position of light
GL_SPOT_DIRECTION	(0.0, 0.0, -1.0)	(x, y, z) direction of spotlight
GL_SPOT_EXPONENT	0.0	spotlight exponent
GL_SPOT_CUTOFF	180.0	spotlight cutoff angle
GL_CONSTANT_ATTENUATION	1.0	constant attenuation factor
GL_LINEAR_ATTENUATION	0.0	linear attenuation factor
GL_QUADRATIC_ATTENUATION	0.0	quadratic attenuation factor

Note: The default values listed for GL_DIFFUSE and GL_SPECULAR in Table 1 apply only to GL_LIGHT(). For other lights, the default value is (0.0, 0.0, 0.0, 1.0) for both GL_DIFFUSE and GL_SPECULAR.

Example 2 shows how to use **glLight*()**:

Example 2 : Defining Colors and Position for a Light Source

```
GLfloat light_ambient[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat light_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

As we can see, arrays are defined for the parameter values, and **glLightfv()** is called repeatedly to set the various parameters. In this example, the first three calls to **glLightfv()** are superfluous, since they're being used to specify the default values for the GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR parameters.

Note: Remember to turn on each light with **glEnable()**.

All the parameters for **glLight*()** and their possible values are explained in the following sections. These parameters interact with those that define the overall lighting model for a particular scene and an object's material properties.

Color

OpenGL allows us to associate three different color-related parameters - GL_AMBIENT, GL_DIFFUSE, and GL_SPECULAR - with any particular light. The GL_AMBIENT parameter refers to the RGBA intensity of the ambient light that a particular light source adds to the scene. As we can see in Table 1, by default there is no ambient light since GL_AMBIENT is (0.0, 0.0, 0.0, 1.0). This value was used in Example 1. If this program had specified blue ambient light as

```
GLfloat light_ambient[] = { 0.0, 0.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
```

The GL_DIFFUSE parameter probably most closely correlates with what we naturally think of as "the color of a light." It defines the RGBA color of the diffuse light that a particular light source adds to a scene. By default, GL_DIFFUSE is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0, which produces a bright. The default value for any other light (GL_LIGHT1, ... , GL_LIGHT7) is (0.0, 0.0, 0.0, 0.0).

The GL_SPECULAR parameter affects the color of the specular highlight on an object. Typically, a real-world object such as a glass bottle has a specular highlight that's the color of the light shining on it (which is often white). Therefore, if we want to create a realistic effect, set the GL_SPECULAR parameter to the same value as the GL_DIFFUSE parameter. By default, GL_SPECULAR is (1.0, 1.0, 1.0, 1.0) for GL_LIGHT0 and (0.0, 0.0, 0.0, 0.0) for any other light.

Note: The alpha component of these colors is not used until blending is enabled.

Position and Attenuation

As previously mentioned, we can choose whether to have a light source that's treated as though it's located infinitely far away from the scene or one that's nearer to the scene. The first type is referred to as a *directional* light source; the effect of an infinite location is that the rays of light can be considered parallel by the time they reach an object. An example of a real-world directional light source is the sun. The second type is called a *positional* light source, since its exact position within the scene determines the effect it has on a scene and, specifically, the direction from which the light rays come. A desk lamp is an example of a positional light source. The light used in Example 1 is a directional one:

```
GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };  
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
```

As shown, we supply a vector of four values (x, y, z, w) for the GL_POSITION parameter. If the last value, w, is zero, the corresponding light source is a directional one, and the (x, y, z) values describe its direction. This direction is transformed by the modelview matrix. By default, GL_POSITION is (0, 0, 1, 0), which defines a directional light that points along the negative z-axis. (Note that nothing prevents us from creating a directional light with the direction of (0, 0, 0), but such a light won't help us much.) If the w value is nonzero, the light is positional, and the (x, y, z) values specify the location of the light in homogeneous object coordinates. This location is transformed by the modelview matrix and stored in eye coordinates. Also, by default, a positional light radiates in all directions, but we can restrict it to producing a cone of illumination by defining the light as a spotlight.

Note: Remember that the colors across the face of a smooth-shaded polygon are determined by the colors calculated for the vertices. Because of this, we probably want to avoid using large polygons with local lights. If we locate the light near the middle of the polygon, the vertices might be too far away to receive much light, and the whole polygon will look darker than we intended. To avoid this problem, break up the large polygon into smaller ones.

For real-world lights, the intensity of light decreases as distance from the light increases. Since a directional light is infinitely far away, it doesn't make sense to attenuate its intensity over distance, so attenuation is disabled for a directional light. However, we might want to attenuate the light from a positional light. OpenGL attenuates a light source by multiplying the contribution of that source by an attenuation factor:

$$\text{attenuation factor} = \frac{1}{k_c + k_l d + k_q d^2}$$

where

d = distance between the light's position and the vertex

k_c = GL_CONSTANT_ATTENUATION

```
k1 = GL_LINEAR_ATTENUATION
```

```
kq = GL_QUADRATIC_ATTENUATION
```

By default, `kc` is 1.0 and both `k1` and `kq` are zero, but we can give these parameters different values:

```
glLightf(GL_LIGHT0, GL_CONSTANT_ATTENUATION, 2.0);
```

```
glLightf(GL_LIGHT0, GL_LINEAR_ATTENUATION, 1.0);
```

```
glLightf(GL_LIGHT0, GL_QUADRATIC_ATTENUATION, 0.5);
```

Note that the ambient, diffuse, and specular contributions are all attenuated. Only the emission and global ambient values aren't attenuated. Also note that since attenuation requires an additional division (and possibly more math) for each calculated color, using attenuated lights may slow down application performance.

Spotlights

As previously mentioned, we can have a positional light source act as a spotlight - that is, by restricting the shape of the light it emits to a cone. To create a spotlight, we need to determine the spread of the cone of light we desire. (Remember that since spotlights are positional lights, we also have to locate them where we want them. Again, note that nothing prevents us from creating a directional spotlight, but it won't give us the result we want.) To specify the angle between the axis of the cone and a ray along the edge of the cone, use the `GL_SPOT_CUTOFF` parameter. The angle of the cone at the apex is then twice this value, as shown in Figure 2.

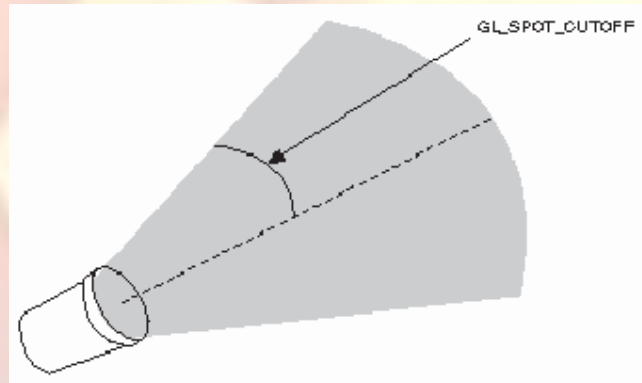


Figure 2 : GL_SPOT_CUTOFF Parameter

Note that no light is emitted beyond the edges of the cone. By default, the spotlight feature is disabled because the `GL_SPOT_CUTOFF` parameter is 180.0. This value means that light is emitted in all directions (the angle at the cone's apex is 360 degrees, so it isn't a cone at all). The value for `GL_SPOT_CUTOFF` is restricted to being within the range `[0.0,90.0]` (unless it has the special value 180.0). The following line sets the cutoff parameter to 45 degrees:

```
glLightf(GL_LIGHT0, GL_SPOT_CUTOFF, 45.0);
```

We also need to specify a spotlight's direction, which determines the axis of the cone of light:

```
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT0, GL_SPOT_DIRECTION, spot_direction);
```

The direction is specified in object coordinates. By default, the direction is (0.0, 0.0, -1.0), so if we don't explicitly set the value of `GL_SPOT_DIRECTION`, the light points down the negative z-axis. Also, keep in mind that a spotlight's direction is transformed by the modelview matrix just as though it were a normal vector, and the result is stored in eye coordinates.

In addition to the spotlight's cutoff angle and direction, there are two ways we can control the intensity distribution of the light within the cone. First, we can set the attenuation factor described earlier, which is multiplied by the light's intensity. We can also set the `GL_SPOT_EXPONENT` parameter, which by default is zero, to control how concentrated the light is. The light's intensity is highest in the center of the cone. It's attenuated toward the edges of the cone by the cosine of the angle between the direction of the light and the direction from the light to the vertex being lit, raised to the power of the spot exponent. Thus, higher spot exponents result in a more focused light source.

Multiple Lights

As mentioned, we can have at least eight lights in our scene (possibly more, depending on our OpenGL implementation). Since OpenGL needs to perform calculations to determine how much light each vertex receives from each light source, increasing the number of lights adversely affects performance. The constants used to refer to the eight lights are `GL_LIGHT0`, `GL_LIGHT1`, `GL_LIGHT2`, `GL_LIGHT3`, and so on. In the preceding discussions, parameters related to `GL_LIGHT0` were set. If we want an additional light, we need to specify its parameters; also, remember that the default values are different for these other lights than they are for `GL_LIGHT0`, as explained in Table 1. Example 3 defines a white attenuated spotlight.

Example 3 : Second Light Source

```
GLfloat light1_ambient[] = { 0.2, 0.2, 0.2, 1.0 };
GLfloat light1_diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat light1_position[] = { -2.0, 2.0, 1.0, 1.0 };
GLfloat spot_direction[] = { -1.0, -1.0, 0.0 };
glLightfv(GL_LIGHT1, GL_AMBIENT, light1_ambient);
glLightfv(GL_LIGHT1, GL_DIFFUSE, light1_diffuse);
glLightfv(GL_LIGHT1, GL_SPECULAR, light1_specular);
glLightfv(GL_LIGHT1, GL_POSITION, light1_position);
glLightf(GL_LIGHT1, GL_CONSTANT_ATTENUATION, 1.5);
glLightf(GL_LIGHT1, GL_LINEAR_ATTENUATION, 0.5);
glLightf(GL_LIGHT1, GL_QUADRATIC_ATTENUATION, 0.2);
glLightf(GL_LIGHT1, GL_SPOT_CUTOFF, 45.0);
glLightfv(GL_LIGHT1, GL_SPOT_DIRECTION, spot_direction);
glLightf(GL_LIGHT1, GL_SPOT_EXPONENT, 2.0);
glEnable(GL_LIGHT1);
```

If these lines were added to Example 1, the sphere would be lit with two lights, one directional and one spotlight.

Try This

Modify Example 1 in the following manner:

Change the first light to be a positional colored light rather than a directional white one. Add an additional colored spotlight. Hint: Use some of the code shown in the preceding section. Measure how these two changes affect performance.

Controlling a Light's Position and Direction

OpenGL treats the position and direction of a light source just as it treats the position of a geometric primitive. In other words, a light source is subject to the same matrix transformations as a primitive. More specifically, when `glLight*()` is called to specify the position or the direction of a light source, the position or direction is transformed by the current modelview matrix and stored in eye coordinates. This means we can manipulate a light source's position or direction by changing the contents of the modelview matrix. (The projection matrix has no effect on a light's position or direction.) This section explains how to achieve the following three different effects by changing the point in the program at which the light position is set, relative to modeling or viewing transformations:

- A light position that remains fixed
- A light that moves around a stationary object
- A light that moves along with the viewpoint

Keeping the Light Stationary

In the simplest example, as in Example 1, the light position remains fixed. To achieve this effect, we need to set the light position after whatever viewing and/or modeling transformation we use. In Example 4, the relevant code from the `init()` and `reshape()` routines might look like this.

Example 4 : Stationary Light Source

```
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho (-1.5, 1.5, -1.5*h/w, 1.5*h/w, -10.0, 10.0);
else
glOrtho (-1.5*w/h, 1.5*w/h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode (GL_MODELVIEW);
glLoadIdentity();
/* later in init() */
GLfloat light_position[] = { 1.0, 1.0, 1.0, 1.0 };
glLightfv(GL_LIGHT0, GL_POSITION, position);
```

As we can see, the viewport and projection matrices are established first. Then, the identity matrix is loaded as the modelview matrix, after which the light position is set. Since the identity matrix is used, the originally specified light position (1.0, 1.0, 1.0) isn't changed by being multiplied by the modelview matrix. Then, since neither the light position nor the modelview matrix is modified after this point, the direction of the light remains (1.0, 1.0, 1.0).

Independently Moving the Light

Now suppose we want to rotate or translate the light position so that the light moves relative to a stationary object. One way to do this is to set the light position after the modeling transformation, which is itself changed specifically to modify the light position. We can begin with the same series of calls in **init()** early in the program. Then we need to perform the desired modeling transformation (on the modelview stack) and reset the light position, probably in **display()**. Example 5 shows what **display()** might be.

Example 5 : Independently Moving Light Source

```
static GLdouble spin;
void display(void)
{
    GLfloat light_position[] = { 0.0, 0.0, 1.5, 1.0 };
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix();
    gluLookAt(0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 1.0, 0.0);
    glPushMatrix();
    glRotated(spin, 1.0, 0.0, 0.0);
    glLightfv(GL_LIGHT0, GL_POSITION, light_position);
    glPopMatrix();
    glutSolidTorus(0.275, 0.85, 8, 15);
    glPopMatrix();
    glFlush();
}
```

`spin` is a global variable and is probably controlled by an input device. **display()** causes the scene to be redrawn with the light rotated `spin` degrees around a stationary torus. Note the two pairs of **glPushMatrix()** and **glPopMatrix()** calls, which are used to isolate the viewing and modeling transformations, all of which occur on the modelview stack. Since in Example 5 the viewpoint remains constant, the current matrix is pushed down the stack and then the desired viewing transformation is loaded with **gluLookAt()**. The matrix stack is pushed again before the modeling transformation **glRotated()** is specified. Then the light position is set in the new, rotated coordinate system so that the light itself appears to be rotated from its previous position. (Remember that the light position is stored in eye coordinates, which are obtained after transformation by the modelview matrix.) After the rotated matrix is popped off the stack, the torus is drawn.

Example 6 is a program that rotates a light source around an object. When the left mouse button is pressed, the light position rotates an additional 30 degrees. A small, unlit, wireframe cube is drawn to represent the position of the light in the scene.

Example 6 : Moving a Light with Modeling Transformations:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
static int spin = 0;
void init(void)
{
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glShadeModel(GL_SMOOTH);
```

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_DEPTH_TEST);
}
/* Here is where the light position is reset after the modeling
 * transformation (glRotated) is called. This places the
 * light at a new position in world coordinates. The cube
 * represents the position of the light.
 */
void display(void)
{
    GLfloat position[] = { 0.0, 0.0, 1.5, 1.0 };
    glClear (GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glPushMatrix ();
    glTranslatef (0.0, 0.0, -5.0);
    glPushMatrix ();
    glRotated ((GLfloat) spin, 1.0, 0.0, 0.0);
    glLightfv (GL_LIGHT0, GL_POSITION, position);
    glTranslated (0.0, 0.0, 1.5);
    glDisable (GL_LIGHTING);
    glColor3f (0.0, 1.0, 1.0);
    glutWireCube (0.1);
    glEnable (GL_LIGHTING);
    glPopMatrix ();
    glutSolidTorus (0.275, 0.85, 8, 15);
    glPopMatrix ();
    glFlush ();
}
void reshape (int w, int h)
{
    glViewport (0, 0, (GLsizei) w, (GLsizei) h);
    glMatrixMode (GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(40.0, (GLfloat) w/(GLfloat) h, 1.0, 20.0);
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
    switch (button) {
    case GLUT_LEFT_BUTTON:
    if (state == GLUT_DOWN) {
    spin = (spin + 30) % 360;
    glutPostRedisplay();
    }
    break;
    default:
    break;
    }
}
```

```

int main(int argc, char** argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize (500, 500);
    glutInitWindowPosition (100, 100);
    glutCreateWindow (argv[0]);
    init ();
    glutDisplayFunc(display);
    glutReshapeFunc(reshape);
    glutMouseFunc(mouse);
    glutMainLoop();
    return 0;
}

```

Selecting a Lighting Model

The OpenGL notion of a lighting model has three components:

The global ambient light intensity

Whether the viewpoint position is local to the scene or whether it should be considered to be an infinite distance away

Whether lighting calculations should be performed differently for both the front and back faces of objects

This section explains how to specify a lighting model. It also discusses how to enable lighting - that is, how to tell OpenGL that we want lighting calculations performed. The command used to specify all properties of the lighting model is **glLightModel*()**. **glLightModel*()** has two arguments: the lighting model property and the desired value for that property.

```
void glLightModel{if}(GLenum pname, TYPEparam);
```

```
void glLightModel{ifv}(GLenum pname, TYPE *param);
```

Sets properties of the lighting model. The characteristic of the lighting model being set is defined by pname, which specifies a named parameter (see Table 2). param indicates the values to which the pname characteristic is set; it's a pointer to a group of values if the vector version is used, or the value itself if the nonvector version is used. The nonvector version can be used to set only single-valued lighting model characteristics, not for GL_LIGHT_MODEL_AMBIENT.

Table 2 : Default Values for pname Parameter of glLightModel*()

Parameter Name	Default Value	Meaning
GL_LIGHT_MODEL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient RGBA intensity of the entire scene
GL_LIGHT_MODEL_LOCAL_VIEWER	0.0 or GL_FALSE	how specular reflection angles are computed
GL_LIGHT_MODEL_TWO_SIDE	0.0 or GL_FALSE	choose between one-sided or two-sided lighting

Global Ambient Light

As discussed earlier, each light source can contribute ambient light to a scene. In addition, there can be other ambient light that's not from any particular source. To specify the RGBA intensity of such global ambient light, use the `GL_LIGHT_MODEL_AMBIENT` parameter as follows:

```
GLfloat lmodel_ambient[] = { 0.2, 0.2, 0.2, 1.0 };  
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodel_ambient);
```

In this example, the values used for `lmodel_ambient` are the default values for `GL_LIGHT_MODEL_AMBIENT`. Since these numbers yield a small amount of white ambient light, even if we don't add a specific light source to our scene, we can still see the objects in the scene.

Enabling Lighting

With OpenGL, we need to explicitly enable (or disable) lighting. If lighting isn't enabled, the current color is simply mapped onto the current vertex, and no calculations concerning normals, light sources, the lighting model, and material properties are performed. Here's how to enable lighting:

```
glEnable(GL_LIGHTING);
```

To disable lighting, call `glDisable()` with `GL_LIGHTING` as the argument. We also need to explicitly enable each light source that we define, after we've specified the parameters for that source. Example 1 uses only one light,

```
GL_LIGHT0:  
glEnable(GL_LIGHT0);
```

Defining Material Properties

We've seen how to create light sources with certain characteristics and how to define the desired lighting model. This section describes how to define the material properties of the objects in the scene: the ambient, diffuse, and specular colors, the shininess, and the color of any emitted light. Most of the material properties are conceptually similar to ones we've already used to create light sources. The mechanism for setting them is similar, except that the command used is called `glMaterial*()`.

```
void glMaterial{if}(GLenum face, GLenum pname, TYPE param);  
void glMaterial{if}v(GLenum face, GLenum pname, TYPE *param);
```

Specifies a current material property for use in lighting calculations. face can be GL_FRONT, GL_BACK, or GL_FRONT_AND_BACK to indicate which face of the object the material should be applied to. The particular material property being set is identified by pname and the desired values for that property are given by param, which is either a pointer to a group of values (if the vector version is used) or the actual value (if the nonvector version is used). The nonvector version works only for setting

GL_SHININESS. The possible values for *pname* are shown in Table 3. Note that *GL_AMBIENT_AND_DIFFUSE* allows us to set both the ambient and diffuse material colors simultaneously to the same RGBA value.

Table 3 : Default Values for *pname* Parameter of `glMaterial*()`

Parameter Name	Default Value	Meaning
GL_AMBIENT	(0.2, 0.2, 0.2, 1.0)	ambient color of material
GL_DIFFUSE	(0.8, 0.8, 0.8, 1.0)	diffuse color of material
GL_AMBIENT_AND_DIFFUSE		ambient and diffuse color of material
GL_SPECULAR	(0.0, 0.0, 0.0, 1.0)	specular color of material
GL_SHININESS	0.0	specular exponent
GL_EMISSION	(0.0, 0.0, 0.0, 1.0)	emissive color of material
GL_COLOR_INDEXES	(0,1,1)	ambient, diffuse, and specular color indices

As discussed in "Selecting a Lighting Model," we can choose to have lighting calculations performed differently for the front- and back-facing polygons of objects. If the back faces might indeed be seen, we can supply different material properties for the front and the back surfaces by using the face parameter of `glMaterial*()`.

Note that most of the material properties set with `glMaterial*()` are (R, G, B, A) colors. Regardless of what alpha values are supplied for other parameters, the alpha value at any particular vertex is the diffuse-material alpha value (that is, the alpha value given to `GL_DIFFUSE` with the `glMaterial*()` command, as described in the next section). Also, none of the RGBA material properties apply in color-index mode.

Diffuse and Ambient Reflection

The `GL_DIFFUSE` and `GL_AMBIENT` parameters set with `glMaterial*()` affect the color of the diffuse and ambient light reflected by an object. Diffuse reflectance plays the most important role in determining what we perceive the color of an object to be. It's affected by the color of the incident diffuse light and the angle of the incident light relative to the normal direction. (It's most intense where the incident light falls perpendicular to the surface.) The position of the viewpoint doesn't affect diffuse reflectance at all.

Ambient reflectance affects the overall color of the object. Because diffuse reflectance is brightest where an object is directly illuminated, ambient reflectance is most noticeable where an object receives no direct illumination. An object's total ambient reflectance is affected by the global ambient light and ambient light from individual light sources. Like

diffuse reflectance, ambient reflectance isn't affected by the position of the viewpoint. For real-world objects, diffuse and ambient reflectance are normally the same color. For this reason, OpenGL provides us with a convenient way of assigning the same value to both simultaneously with `glMaterial*()`:

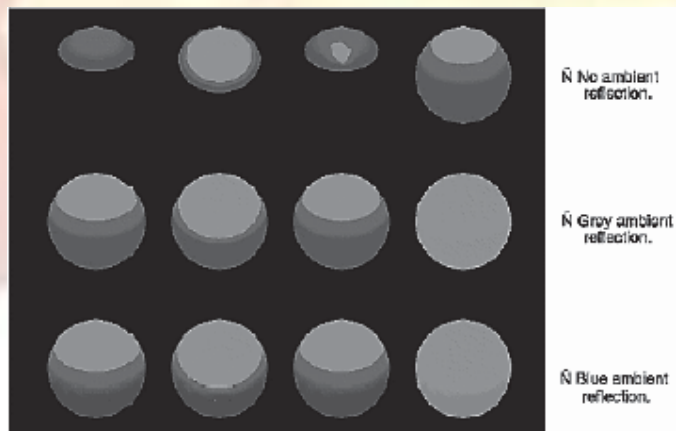
```
GLfloat mat_amb_diff[] = { 0.1, 0.5, 0.8, 1.0 };
glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT_AND_DIFFUSE, mat_amb_diff);
```

In this example, the RGBA color (0.1, 0.5, 0.8, 1.0) - a deep blue color - represents the current ambient and diffuse reflectance for both the front- and back-facing polygons.

Specular Reflection

Specular reflection from an object produces highlights. Unlike ambient and diffuse reflection, the amount of specular reflection seen by a viewer does depend on the location of the viewpoint - it's brightest along the direct angle of reflection. To see this, imagine looking at a metallic ball outdoors in the sunlight. As we move our head, the highlight created by the sunlight moves with us to some extent. However, if we move our head too much, we lose the highlight entirely.

OpenGL allows us to set the effect that the material has on reflected light (with `GL_SPECULAR`) and control the size and brightness of the highlight (with `GL_SHININESS`). We can assign a number in the range of [0.0, 128.0] to `GL_SHININESS` - the higher the value, the smaller and brighter (more focused) the highlight.



Twelve spheres, each with different material parameters. The row properties are as labeled above. The first column uses a blue diffuse material color with no specular properties. The second column adds white specular reflection with a low shininess exponent. The third column uses a high shininess exponent and thus has a more concentrated highlight. The fourth column uses the blue diffuse color and, instead of specular reflection, adds an emissive component.

In above figure, the spheres in the first column have no specular reflection. In the second column, `GL_SPECULAR` and `GL_SHININESS` are assigned values as follows:

```

GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat low_shininess[] = { 5.0 };
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);

```

In the third column, the GL_SHININESS parameter is increased to 100.0.

Emission

By specifying an RGBA color for GL_EMISSION, we can make an object appear to be giving off light of that color. Since most real-world objects (except lights) don't emit light, we'll probably use this feature mostly to simulate lamps and other light sources in a scene.

```

GLfloat mat_emission[] = { 0.3, 0.2, 0.2, 0.0 };
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);

```

Notice that the spheres appear to be slightly glowing; however, they're not actually acting as light sources. We would need to create a light source and position it at the same location as the sphere to create that effect.

Changing Material Properties

Example 1 uses the same material properties for all vertices of the only object in the scene (the sphere). In other situations, we might want to assign different material properties for different vertices on the same object. More likely, we have more than one object in the scene, and each object has different material properties. For example, the code that produced "above figure" has to draw twelve different objects (all spheres), each with different material properties. Example 8 shows a portion of the code in **display()**.

Example 8 : Different Material Properties:

```

GLfloat no_mat[] = { 0.0, 0.0, 0.0, 1.0 };
GLfloat mat_ambient[] = { 0.7, 0.7, 0.7, 1.0 };
GLfloat mat_ambient_color[] = { 0.8, 0.8, 0.2, 1.0 };
GLfloat mat_diffuse[] = { 0.1, 0.5, 0.8, 1.0 };
GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
GLfloat no_shininess[] = { 0.0 };
GLfloat low_shininess[] = { 5.0 };
GLfloat high_shininess[] = { 100.0 };
GLfloat mat_emission[] = { 0.3, 0.2, 0.2, 0.0 };
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
/* draw sphere in first row, first column
 * diffuse reflection only; no ambient or specular
 */
glPushMatrix();
glTranslatef (-3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);

```

```

glutSolidSphere(1.0, 16, 16);
glPopMatrix();
/* draw sphere in first row, second column
 * diffuse and specular reflection; low shininess; no ambient
 */
glPushMatrix();
glTranslatef (-1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, low_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();
/* draw sphere in first row, third column
 * diffuse and specular reflection; high shininess; no ambient
 */
glPushMatrix();
glTranslatef (1.25, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialfv(GL_FRONT, GL_SHININESS, high_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, no_mat);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();
/* draw sphere in first row, fourth column
 * diffuse reflection; emission; no ambient or specular refl.
 */
glPushMatrix();
glTranslatef (3.75, 3.0, 0.0);
glMaterialfv(GL_FRONT, GL_AMBIENT, no_mat);
glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR, no_mat);
glMaterialfv(GL_FRONT, GL_SHININESS, no_shininess);
glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
glutSolidSphere(1.0, 16, 16);
glPopMatrix();

```

As we can see, **glMaterialfv()** is called repeatedly to set the desired material property for each sphere. Note that it only needs to be called to change a property that needs to be respecified. The second, third, and fourth spheres use the same ambient and diffuse properties as the first sphere, so these properties do not need to be respecified. Since **glMaterial*()** has a performance cost associated with its use, Example 8 could be rewritten to minimize material-property changes. Another technique for minimizing performance costs associated with changing material properties is to use **glColorMaterial()**.

```
void glColorMaterial(GLenum face, GLenum mode);
```

Causes the material property (or properties) specified by mode of the specified material face (or faces) specified by face to track the value of the current color at all times. A change to the current color (using **glColor*()**) immediately updates the specified material properties. The face parameter can be `GL_FRONT`, `GL_BACK`, or `GL_FRONT_AND_BACK` (the default). The mode parameter can be `GL_AMBIENT`, `GL_DIFFUSE`, `GL_AMBIENT_AND_DIFFUSE` (the default), `GL_SPECULAR`, or `GL_EMISSION`. At any given time, only one mode is active. **glColorMaterial()** has no effect on color-index lighting.

Note that **glColorMaterial()** specifies two independent values: the first specifies which face or faces are updated, and the second specifies which material property or properties of those faces are updated. OpenGL does *not* maintain separate mode variables for each face. After calling **glColorMaterial()**, we need to call **glEnable()** with `GL_COLOR_MATERIAL` as the parameter. Then, we can change the current color using **glColor*()** (or other material properties, using **glMaterial*()**) as needed as we draw:

```
glEnable(GL_COLOR_MATERIAL);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
/* now glColor* changes diffuse reflection */
glColor3f(0.2, 0.5, 0.8);
/* draw some objects here */
glColorMaterial(GL_FRONT, GL_SPECULAR);
/* glColor* no longer changes diffuse reflection */
/* now glColor* changes specular reflection */
glColor3f(0.9, 0.0, 0.2);
/* draw other objects here */
glDisable(GL_COLOR_MATERIAL);
```

We should use **glColorMaterial()** whenever we need to change a single material parameter for most vertices in our scene. If we need to change more than one material parameter, as was the case for "in above figure", use **glMaterial*()**. When we don't need the capabilities of **glColorMaterial()** anymore, be sure to disable it so that we don't get undesired material properties and don't incur the performance cost associated with it. The performance value in using **glColorMaterial()** varies, depending on our OpenGL implementation. Some implementations may be able to optimize the vertex routines so that they can quickly update material properties based on the current color.

Example 9 shows an interactive program that uses **glColorMaterial()** to change material parameters. Pressing each of the three mouse buttons changes the color of the diffuse reflection.

Example 9 : Using **glColorMaterial()**:

```
#include <GL/gl.h>
#include <GL/glu.h>
#include "glut.h"
GLfloat diffuseMaterial[4] = { 0.5, 0.5, 0.5, 1.0 };
void init(void)
{
    GLfloat mat_specular[] = { 1.0, 1.0, 1.0, 1.0 };
    GLfloat light_position[] = { 1.0, 1.0, 1.0, 0.0 };
```

```
glClearColor (0.0, 0.0, 0.0, 0.0);
glShadeModel (GL_SMOOTH);
glEnable(GL_DEPTH_TEST);
glMaterialfv(GL_FRONT, GL_DIFFUSE, diffuseMaterial);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
glMaterialf(GL_FRONT, GL_SHININESS, 25.0);
glLightfv(GL_LIGHT0, GL_POSITION, light_position);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glColorMaterial(GL_FRONT, GL_DIFFUSE);
glEnable(GL_COLOR_MATERIAL);
}
void display(void)
{
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
glutSolidSphere(1.0, 20, 16);
glFlush ();
}
void reshape (int w, int h)
{
glViewport (0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode (GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho (-1.5, 1.5, -1.5*(GLfloat)h/(GLfloat)w, 1.5*(GLfloat)h/(GLfloat)w, -10.0, 10.0);
else
glOrtho (-1.5*(GLfloat)w/(GLfloat)h, 1.5*(GLfloat)w/(GLfloat)h, -1.5, 1.5, -10.0, 10.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}
void mouse(int button, int state, int x, int y)
{
switch (button) {
case GLUT_LEFT_BUTTON:
if (state == GLUT_DOWN) { /* change red */
diffuseMaterial[0] += 0.1;
if (diffuseMaterial[0] > 1.0)
diffuseMaterial[0] = 0.0;
glColor4fv(diffuseMaterial);
glutPostRedisplay();
}
break;
case GLUT_MIDDLE_BUTTON:
if (state == GLUT_DOWN) { /* change green */
diffuseMaterial[1] += 0.1;
if (diffuseMaterial[1] > 1.0)
diffuseMaterial[1] = 0.0;
glColor4fv(diffuseMaterial);
glutPostRedisplay();
}
}
```

```

break;
case GLUT_RIGHT_BUTTON:
if (state == GLUT_DOWN) { /* change blue */
diffuseMaterial[2] += 0.1;
if (diffuseMaterial[2] > 1.0)
diffuseMaterial[2] = 0.0;
glColor4fv(diffuseMaterial);
glutPostRedisplay();
}
break;
default:
break;
}
}
int main(int argc, char** argv)
{
glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB | GLUT_DEPTH);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMouseFunc(mouse);
glutMainLoop();
return 0;
}

```

Try This

Modify Example 8 in the following manner:

Change the global ambient light in the scene. Hint: Alter the value of the `GL_LIGHT_MODEL_AMBIENT` parameter.

Change the diffuse, ambient, and specular reflection parameters, the shininess exponent, and the emission color. Hint: Use the `glMaterial*()` command, but avoid making excessive calls.

Use two-sided materials and add a user-defined clipping plane so that we can see the inside and outside of a row or column of spheres. (if we need to recall user-defined clipping planes.) Hint: Turn on two-sided lighting with `GL_LIGHT_MODEL_TWO_SIDE`, set the desired material properties, and add a clipping plane.

Remove all the `glMaterialfv()` calls, and use the more efficient `glColorMaterial()` calls to achieve the same lighting.

Lecture No.44 Evaluators, curves and Surfaces

Evaluators

A Bézier curve is a vector-valued function of one variable

$$C(u) = [X(u) \ Y(u) \ Z(u)]$$

where u varies in some domain (say $[0,1]$).

A Bézier surface patch is a vector-valued function of two variables

$$S(u,v) = [X(u,v) \ Y(u,v) \ Z(u,v)]$$

Where u and v can both vary in some domain. The range isn't necessarily three-dimensional as shown here. You might want two-dimensional output for curves on a plane or texture coordinates, or you might want four-dimensional output to specify RGBA information. Even one-dimensional output may make sense for gray levels. For each u (or u and v , in the case of a surface), the formula for $C()$ (or $S()$) calculates a point on the curve (or surface). To use an evaluator, first define the function $C()$ or $S()$, enable it, and then use the `glEvalCoord1()` or `glEvalCoord2()` command instead of `glVertex*()`. This way, the curve or surface vertices can be used like any other vertices - to form points or lines, for example. In addition, other commands automatically generate series of vertices that produce a regular mesh uniformly spaced in u (or in u and v). One- and two-dimensional evaluators are similar, but the description is somewhat simpler in one dimension, so that case is discussed first.

One-Dimensional Evaluators

This section presents an example of using one-dimensional evaluators to draw a curve. It then describes the commands and equations that control evaluators.

One-Dimensional Example: A Simple Bézier Curve

The program shown in Example 1 draws a cubic Bézier curve using four control points, as shown in Figure 1.

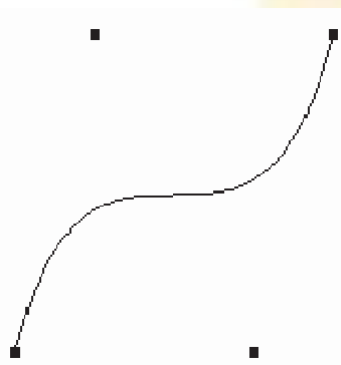


Figure 1 : Bézier Curve

Example 1 : Bézier Curve with Four Control Points:

```

#include <GL/gl.h>
#include <GL/glu.h>
#include <stdlib.h>
#include <GL/glut.h>
GLfloat ctrlpnts[4][3] = {
{-4.0, -4.0, 0.0}, {-2.0, 4.0, 0.0},
{2.0, -4.0, 0.0}, {4.0, 4.0, 0.0}};
void init(void)
{
glClearColor(0.0, 0.0, 0.0, 0.0);
glShadeModel(GL_FLAT);
glMap1f(GL_MAP1_VERTEX_3, 0.0, 1.0, 3, 4, &ctrlpnts[0][0]);
glEnable(GL_MAP1_VERTEX_3);
}
void display(void)
{
int i;
glClear(GL_COLOR_BUFFER_BIT);
glColor3f(1.0, 1.0, 1.0);
glBegin(GL_LINE_STRIP);
for (i = 0; i <= 30; i++)
glEvalCoord1f((GLfloat) i/30.0);
glEnd();
/* The following code displays the control points as dots. */
glPointSize(5.0);
glColor3f(1.0, 1.0, 0.0);
glBegin(GL_POINTS);
for (i = 0; i < 4; i++)
glVertex3fv(&ctrlpnts[i][0]);
glEnd();
glFlush();
}
void reshape(int w, int h)
{
glViewport(0, 0, (GLsizei) w, (GLsizei) h);
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
if (w <= h)
glOrtho(-5.0, 5.0, -5.0*(GLfloat)h/(GLfloat)w,
5.0*(GLfloat)h/(GLfloat)w, -5.0, 5.0);
else
glOrtho(-5.0*(GLfloat)w/(GLfloat)h,
5.0*(GLfloat)w/(GLfloat)h, -5.0, 5.0, -5.0, 5.0);
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
}

int main(int argc, char** argv)
{

```

```

glutInit(&argc, argv);
glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
glutInitWindowSize (500, 500);
glutInitWindowPosition (100, 100);
glutCreateWindow (argv[0]);
init ();
glutDisplayFunc(display);
glutReshapeFunc(reshape);
glutMainLoop();
return 0;
}

```

A cubic Bézier curve is described by four control points, which appear in this example in the `ctrlpoints[][]` array. This array is one of the arguments to `glMap1f()`. All the arguments for this command are as follows:

`GL_MAP1_VERTEX_3`

Three-dimensional control points are provided and three-dimensional vertices are produced

0.0 : Low value of parameter `u`

1.0 : High value of parameter `u`

3 : The number of floating-point values to advance in the data between one control point and the next

4 : The order of the spline, which is the degree+1: in this case, the degree is 3 (since this is a cubic curve)

&ctrlpoints[0][0] : Pointer to the first control point's data

Note that the second and third arguments control the parameterization of the curve - as the variable `u` ranges from 0.0 to 1.0, the curve goes from one end to the other. The call to `glEnable()` enables the one-dimensional evaluator for three-dimensional vertices.

The curve is drawn in the routine `display()` between the `glBegin()` and `glEnd()` calls. Since the evaluator is enabled, the command `glEvalCoord1f()` is just like issuing a `glVertex()` command with the coordinates of a vertex on the curve corresponding to the input parameter `u`.

44.1 Defining and Evaluating a One-Dimensional Evaluator

The Bernstein polynomial of degree `n` (or order `n+1`) is given by

$$B_i^n(u) = \binom{n}{i} u^i (1-u)^{n-i}$$

If `Pi` represents a set of control points (one-, two-, three-, or even four- dimensional), then the equation

$$C(u) = \sum_{i=0}^n B_i^n(u) P_i$$

represents a Bézier curve as u varies from 0.0 to 1.0. To represent the same curve but allowing u to vary between u_1 and u_2 instead of 0.0 and 1.0, evaluate

$$C\left(\frac{u - u_1}{u_2 - u_1}\right)$$

The command **glMap1()** defines a one-dimensional evaluator that uses these equations.

```
void glMap1{fd}(GLenum target, TYPEu1, TYPEu2, GLint stride,
GLint order, const TYPE*points);
```

Defines a one-dimensional evaluator. The target parameter specifies what the control points represent, as shown in Table 1, and therefore how many values need to be supplied in points. The points can represent vertices, RGBA color data, normal vectors, or texture coordinates. For example, with GL_MAP1_COLOR_4, the evaluator generates color data along a curve in four-dimensional (RGBA) color space. You also use the parameter values listed in Table 1 to enable each defined evaluator before you invoke it. Pass the appropriate value to glEnable() or glDisable() to enable or disable the evaluator. The second two parameters for glMap1(), u1 and u2, indicate the range for the variable u. The variable stride is the number of single- or double-precision values (as appropriate) in each block of storage. Thus, it's an offset value between the beginning of one control point and the beginning of the next. The order is the degree plus one, and it should agree with the number of control points. The points parameter points to the first coordinate of the first control point. Using the example data structure for glMap1*(), use the following for points:*

```
(GLfloat *)(&ctlpoints[0].x)
```

Table 1 : Types of Control Points for glMap1*()

Parameter	Meaning
GL_MAP1_VERTEX_3	x, y, z vertex coordinates
GL_MAP1_VERTEX_4	x, y, z, w vertex coordinates
GL_MAP1_INDEX	color index
GL_MAP1_COLOR_4	R, G, B, A
GL_MAP1_NORMAL	normal coordinates
GL_MAP1_TEXTURE_COORD_1	s texture coordinates
GL_MAP1_TEXTURE_COORD_2	s, t texture coordinates
GL_MAP1_TEXTURE_COORD_3	s, t, r texture coordinates
GL_MAP1_TEXTURE_COORD_4	s, t, r, q texture coordinates

More than one evaluator can be evaluated at a time. If you have both a `GL_MAP1_VERTEX_3` and a `GL_MAP1_COLOR_4` evaluator defined and enabled, for example, then calls to `glEvalCoord1()` generate both a position and a color. Only one of the vertex evaluators can be enabled at a time, although you might have defined both of them. Similarly, only one of the texture evaluators can be active. Other than that, however, evaluators can be used to generate any combination of vertex, normal, color, and texture-coordinate data. If more than one evaluator of the same type is defined and enabled, the one of highest dimension is used. Use `glEvalCoord1*()` to evaluate a defined and enabled one-dimensional map.

```
void glEvalCoord1{fd}(TYPE u);
void glEvalCoord1{fd}v(TYPE *u);
```

*Causes evaluation of the enabled one-dimensional maps. The argument *u* is the value (or a pointer to the value, in the vector version of the command) of the domain coordinate.*

For evaluated vertices, values for color, color index, normal vectors, and texture coordinates are generated by evaluation. Calls to `glEvalCoord*()` do not use the current values for color, color index, normal vectors, and texture coordinates. `glEvalCoord*()` also leaves those values unchanged.

44.2 Defining Evenly Spaced Coordinate Values in One Dimension

You can use `glEvalCoord1()` with any values for *u*, but by far the most common use is with evenly spaced values, as shown previously in Example 1. To obtain evenly spaced

values, define a one-dimensional grid using **glMapGrid1*()** and then apply it using **glEvalMesh1()**.

```
void glMapGrid1{fd}(GLint n, TYPEu1, TYPEu2);
```

Defines a grid that goes from u1 to u2 in n steps, which are evenly spaced.

```
void glEvalMesh1(GLenum mode, GLint p1, GLint p2);
```

*Applies the currently defined map grid to all enabled evaluators. The mode can be either GL_POINT or GL_LINE, depending on whether you want to draw points or a connected line along the curve. The call has exactly the same effect as issuing a **glEvalCoord1()** for each of the steps between and including p1 and p2, where $0 \leq p1, p2 \leq n$. Programmatically, it's equivalent to the following:*

```
glBegin(GL_POINTS); /* OR glBegin(GL_LINE_STRIP); */
```

```
for (i = p1; i <= p2; i++)
```

```
glEvalCoord1(u1 + i*(u2-u1)/n);
```

```
glEnd();
```

*except that if $i = 0$ or $i = n$, then **glEvalCoord1()** is called with exactly u1 or u2 as its parameter.*

Two-Dimensional Evaluators

In two dimensions, everything is similar to the one-dimensional case, except that all the commands must take two parameters, u and v, into account. Points, colors, normals, or texture coordinates must be supplied over a surface instead of a curve. Mathematically, the definition of a Bézier surface patch is given by

$$S(u, v) = \sum_{i=0}^n \sum_{j=0}^m B_i^n B_j^m (v) P_{ij}$$

where P_{ij} are a set of $m*n$ control points, and the B_i are the same Bernstein polynomials for one dimension. As before, the P_{ij} can represent vertices, normals, colors, or texture coordinates.

The procedure to use two-dimensional evaluators is similar to the procedure for one dimension.

1. Define the evaluator(s) with **glMap2*()**.
2. Enable them by passing the appropriate value to **glEnable()**.
3. Invoke them either by calling **glEvalCoord2()** between a **glBegin()** and **glEnd()** pair or by specifying and then applying a mesh with **glMapGrid2()** and **glEvalMesh2()**.

44.3 Defining and Evaluating a Two-Dimensional Evaluator

Use **glMap2*()** and **glEvalCoord2*()** to define and then invoke a two-dimensional evaluator.

```
void glMap2{fd}(GLenum target, TYPEu1, TYPEu2, GLint ustride,
GLint uorder, TYPEv1, TYPEv2, GLint vstride,
GLint vorder, TYPE points);
```

The target parameter can have any of the values in Table 1, except that the string MAP1 is replaced with MAP2. As before, these values are also used with `glEnable()` to enable the corresponding evaluator. Minimum and maximum values for both u and v are provided as $u1$, $u2$, $v1$, and $v2$. The parameters $ustride$ and $vstride$ indicate the number of single- or double-precision values (as appropriate) between independent settings for these values, allowing users to select a subrectangle of control points out of a much larger array. For example, if the data appears in the form

```
GLfloat ctpoints[100][100][3];
```

and you want to use the 4x4 subset beginning at `ctpoints[20][30]`, choose $ustride$ to be $100*3$ and $vstride$ to be 3. The starting point, `points`, should be set to `&ctpoints[20][30][0]`. Finally, the order parameters, $uorder$ and $vorder$, can be different, allowing patches that are cubic in one direction and quadratic in the other, for example.

```
void glEvalCoord2{fd}(TYPE u, TYPE v);
void glEvalCoord2{fd}v(TYPE *values);
```

Causes evaluation of the enabled two-dimensional maps. The arguments u and v are the values (or a pointer to the values u and v , in the vector version of the command) for the domain coordinates. If either of the vertex evaluators is enabled (`GL_MAP2_VERTEX_3` or `GL_MAP2_VERTEX_4`), then the normal to the surface is computed analytically. This normal is associated with the generated vertex if automatic normal generation has been enabled by passing `GL_AUTO_NORMAL` to `glEnable()`. If it's disabled, the corresponding enabled normal map is used to produce a normal. If no such map exists, the current normal is used.

44.4 Two-Dimensional Example: A Bézier Surface

Example 2 draws a wire frame Bézier surface using evaluators, as shown in Figure 2. In this example, the surface is drawn with nine curved lines in each direction. Each curve is drawn as 30 segments. To get the whole program, add the `reshape()` and `main()` routines from Example 1.

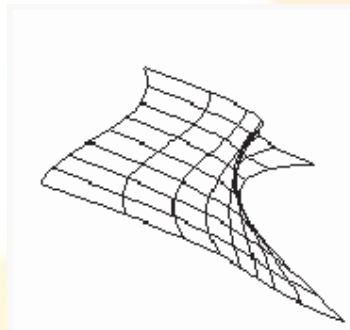


Figure 2 : Bézier Surface

Example 2 : Bézier Surface:

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

```

#include <GL/glut.h>
GLfloat ctrlpoints[4][4][3] = {
    {{-1.5, -1.5, 4.0}, {-0.5, -1.5, 2.0},
     {0.5, -1.5, -1.0}, {1.5, -1.5, 2.0}},
    {{-1.5, -0.5, 1.0}, {-0.5, -0.5, 3.0},
     {0.5, -0.5, 0.0}, {1.5, -0.5, -1.0}},
    {{-1.5, 0.5, 4.0}, {-0.5, 0.5, 0.0},
     {0.5, 0.5, 3.0}, {1.5, 0.5, 4.0}},
    {{-1.5, 1.5, -2.0}, {-0.5, 1.5, -2.0},
     {0.5, 1.5, 0.0}, {1.5, 1.5, -1.0}}
};
void display(void)
{
    int i, j;
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glPushMatrix ();
    glRotatef(85.0, 1.0, 1.0, 1.0);
    for (j = 0; j <= 8; j++) {
        glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)i/30.0, (GLfloat)j/8.0);
        glEnd();
        glBegin(GL_LINE_STRIP);
        for (i = 0; i <= 30; i++)
            glEvalCoord2f((GLfloat)j/8.0, (GLfloat)i/30.0);
        glEnd();
    }
    glPopMatrix ();
    glFlush();
}
void init(void)
{
    glClearColor (0.0, 0.0, 0.0, 0.0);
    glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4.0, 1, 12, 4, &ctrlpoints[0][0][0]);
    glEnable(GL_MAP2_VERTEX_3);
    glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
    glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
}

```

Defining Evenly Spaced Coordinate Values in Two Dimensions

In two dimensions, the **glMapGrid2*()** and **glEvalMesh2()** commands are similar to the one-dimensional versions, except that both u and v information must be included.

```

void glMapGrid2{fd}(GLint nu, TYPEu1, TYPEu2, GLint nv, TYPEv1, TYPEv2);
void glEvalMesh2(GLenum mode, GLint i1, GLint i2, GLint j1, GLint j2);

```

Defines a two-dimensional map grid that goes from $u1$ to $u2$ in nu evenly spaced steps, from $v1$ to $v2$ in nv steps (**glMapGrid2***()), and then applies this grid to all enabled evaluators (**glEvalMesh2***()). The only significant difference from the one-dimensional versions of these two commands is that in **glEvalMesh2**() the mode parameter can be **GL_FILL** as well as **GL_POINT** or **GL_LINE**. **GL_FILL** generates filled polygons using the quad-mesh primitive. Stated precisely, **glEvalMesh2**() is nearly equivalent to one of the following three code fragments. (It's nearly equivalent because when i is equal to nu or j to nv , the parameter is exactly equal to $u2$ or $v2$, not to $u1 + nu*(u2 - u1)/nu$, which might be slightly different due to round-off error.)

```
glBegin(GL_POINTS); /* mode == GL_POINT */
    for (i = nu1; i <= nu2; i++)
        for (j = nv1; j <= nv2; j++)
            glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
glEnd();
```

or

```
for (i = nu1; i <= nu2; i++) { /* mode == GL_LINE */
    glBegin(GL_LINES);
    for (j = nv1; j <= nv2; j++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
for (j = nv1; j <= nv2; j++) {
    glBegin(GL_LINES);
    for (i = nu1; i <= nu2; i++)
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    glEnd();
}
```

or

```
for (i = nu1; i < nu2; i++) { /* mode == GL_FILL */
    glBegin(GL_QUAD_STRIP);
    for (j = nv1; j <= nv2; j++) {
        glEvalCoord2(u1 + i*(u2-u1)/nu, v1+j*(v2-v1)/nv);
        glEvalCoord2(u1 + (i+1)*(u2-u1)/nu, v1+j*(v2-v1)/nv);
    }
    glEnd();
}
```

Example 3 shows the differences necessary to draw the same Bézier surface as Example 2, but using **glMapGrid2**() and **glEvalMesh2**() to subdivide the square domain into a uniform 8x8 grid. This program also adds lighting and shading, as shown in Figure 3.

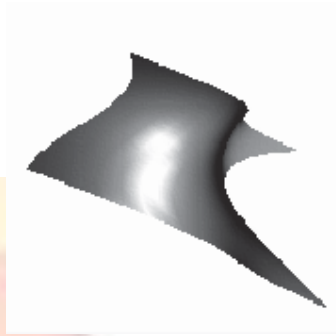


Figure 3 : Lit, Shaded Bézier Surface Drawn with a Mesh

Example 3 : Lit, Shaded Bézier Surface Using a Mesh:

```

void initlights(void)
{
  GLfloat ambient[] = {0.2, 0.2, 0.2, 1.0};
  GLfloat position[] = {0.0, 0.0, 2.0, 1.0};
  GLfloat mat_diffuse[] = {0.6, 0.6, 0.6, 1.0};
  GLfloat mat_specular[] = {1.0, 1.0, 1.0, 1.0};
  GLfloat mat_shininess[] = {50.0};
  glEnable(GL_LIGHTING);
  glEnable(GL_LIGHT0);
  glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
  glLightfv(GL_LIGHT0, GL_POSITION, position);
  glMaterialfv(GL_FRONT, GL_DIFFUSE, mat_diffuse);
  glMaterialfv(GL_FRONT, GL_SPECULAR, mat_specular);
  glMaterialfv(GL_FRONT, GL_SHININESS, mat_shininess);
}

void display(void)
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glPushMatrix();
  glRotatef(85.0, 1.0, 1.0, 1.0);
  glEvalMesh2(GL_FILL, 0, 20, 0, 20);
  glPopMatrix();
  glFlush();
}

void init(void)
{
  glClearColor(0.0, 0.0, 0.0, 0.0);
  glEnable(GL_DEPTH_TEST);
  glMap2f(GL_MAP2_VERTEX_3, 0, 1, 3, 4,
  0, 1, 12, 4, &ctrlpoints[0][0][0]);
  glEnable(GL_MAP2_VERTEX_3);
  glEnable(GL_AUTO_NORMAL);
  glMapGrid2f(20, 0.0, 1.0, 20, 0.0, 1.0);
  initlights();
}

```

Lecture No.45 Animations

The passage of time has fascinated artists, scientists and theologians for thousands of years. Naturally they have attributed to it different interpretations, different implications and different conclusions. Nevertheless there seems to be general agreement on one aspect of time; that we are all conditioned by it and that, whether we like it or not, there is a time space into which we inevitably have to fit.

Einstein, among other well known names in the world of science, made a special study of time in relation to his research in physics. His theory of relativity maintains that space and time are merely different aspects of the same thing. Since then other physicists have pointed out that objects can be moved backward and forward in space, but nothing can be moved back in time.

Another method of describing the concept of time is through the 'three arrows of time'. The first 'arrow' is thermodynamic and can be seen operating when sugar dissolves in hot water. Second, is the historical 'arrow', whereby a single-celled organism evolves to produce more complex and varied species? The third is the cosmological 'arrow' which is the theory that the universe is expanding from a 'big bang' in the past. This cosmic expansion cannot be reversed in time. While the principle of relationships between the 'time arrows' is still to be worked out on a scientific level, the actual application of it is constantly related to all work which utilizes it, such as music and the performing arts. In the latter it is one of the most important raw materials.

In terms of animation, the idea of film time is one of the most vital concepts to understand and to use. It is an essential raw material which can be compressed or expanded and used for effects and moods in a highly creative way. It is, therefore, essential to learn and to understand how time can be applied to animation. The great advantage of animation is that the animator can creatively manipulate time since an action must be timed prior to carrying out the actual physical work on a film.

It is also essential to understand how the audience will react to the manipulation of time from their point of view. Time sense or 'a sense of timing', therefore, is just as important as color sense and skill of drawing or craftsmanship in film animation.

It has to be realized that while a performance on stage and on the screen requires a basic understanding of how timing works, this lecture is primarily confined to hand drawn animation which up to this point in film history still comprises 90% of all output in the animation medium.

Timing for TV series

For economic reasons, TV series are made as simply as possible from the animation point of view. This approach is generally known as limited animation. Animation is expensive, non-animation is cheaper. So to keep the films lively the plots are usually carried along by means of dialogue. It is often necessary to work with prerecorded blocks of dialogue which must remain intact. If this dialogue is well recorded for maximum dramatic effect, lengths of pauses between phrases

cannot be changed (except within very narrow limits) without destroying that effect. In this case, the overall timing of long sections of the film is governed entirely by the dialogue. (There could be, however, considerable flexibility for more detailed timing within this fixed overall length.)

The director has room to maneuver sections. So, if the total timing for all the recorded dialogue is subtracted from the required length for the whole film, this gives the amount of time that is available without dialogue. This can then be split up in the normal way and distributed throughout the film to give the best effect.

Limited animation

With limited animation as many repeats as possible are used within the 24 frames per second. A hold is also lengthened to reduce the number of drawings. As a rule not more than 6 drawings are produced for one second of animation. Limited animation requires almost as much skill on the part of the animator as full animation, since he must create an illusion of action with the greatest sense of economy.

Full animation

Full animation implies a large number of drawings per second of action. Some action may require that every single frame of the 24 frames within the second is animated in order to achieve an illusion of fluidity on the screen. Neither time nor money is spared on animation. As a rule only, TV commercials and feature-length animated films can afford this luxury.

Animation is expensive and time-consuming. It is not economically possible to animate more than is needed and edit the scenes later, as it is in live-action films. In cartoons the director carefully pre-times every action so that the animator works within exact limits and does no more drawings than necessary.

Ideally, the director should be able to view line test loops of the film as it progresses and so have a chance to make adjustments. But often there is no time to make corrections in limited animation and the aim is to make the animation work the first time.

Timing for Animation in general

Timing in animation is an elusive subject. It only exists whilst the film is being projected, in the same way that a melody only exists when it is being played. A melody is more easily appreciated by listening to it than by trying to explain it in words. So with cartoon timing, it is difficult to avoid using a lot of words to explain what may seem fairly simple when seen on the screen.

Timing is also a dangerous factor to try to formulate—something which works in one situation or in one mood may not work at all in another situation or mood. The only real criterion for timing is: if it works effectively on the screen it is good, if it doesn't, it isn't.

So if having looked through the following pages you can see a better way to achieve an effect, then go ahead and do it!

In this book we attempt to look at the laws of movement in nature. What do movements mean? What do they express? How can these movements be

simplified and exaggerated to be made ‘animatable’ and to express ideas, feelings and dramatic effects? The timing mainly described is that which is used in so-called ‘classical’ or ‘full’ animation. To cover all possible kinds of timing in all possible kinds of animation would be quite impossible.



Nevertheless we hope to provide a basic understanding of how timing in animation is ultimately based on timing in nature and how, from this starting point, it is possible to apply such a difficult and invisible concept to the maximum advantage in film animation.

Animation Principles

What is good timing?

Timing is the part of animation which gives *meaning* to movement. Movement can easily be achieved by drawing the same thing in two different positions and inserting a number of other drawings between the two. The result on the screen will be movement, but it will not be animation. In nature, things do not just *move*. Newton's first law of motion stated that things do not move unless a force acts upon them. So in animation the movement itself is of secondary importance; the vital factor is how the action expresses the underlying causes of the movement. With inanimate objects these causes may be natural forces, mainly gravity. With living characters the same external forces can cause movement, plus the contractions of muscles but, more importantly, there are the underlying will, mood, instincts and so on of the character who is moving.

In order to animate a character from A to B, the forces which are operating to produce the movement must be considered. Firstly, gravity tends to pull the character down towards the ground. Secondly, his body is built and jointed in a certain way and is acted on by a certain arrangement of muscles which tend to work against gravity. Thirdly, there is the psychological reason or motivation for his action—whether he is dodging a blow, welcoming a guest or threatening someone with a revolver.

A live actor faced with these problems moves his muscles and limbs and deals with gravity automatically from habit, and so can concentrate on acting. An animator has to worry about making his flat, weightless drawings move like solid, heavy objects, as well as making them act in a convincing way. In both these aspects of animation, timing is of primary importance.

Part of the working storyboard of *The Story of the Bible* by Halas and Batchelor. At this stage the director works out the smooth visual flow of the film, the editing,

camera movements and so on. All these elements combine to tell the story in an interesting way.

The storyboard

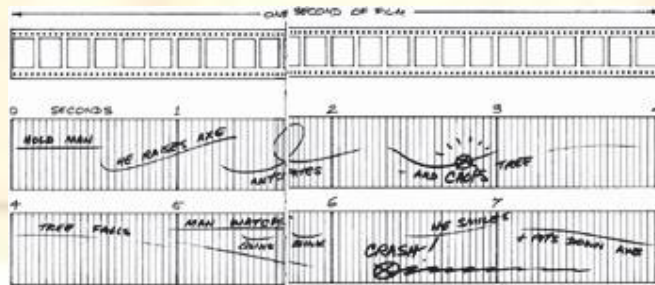
A smooth visual flow is the major objective in any film, especially if it is an animated one. Good continuity depends on coordinating the action of the character, choreography, scene changes and camera movement. All these different aspects cannot be considered in isolation. They must work together to put across a story point. Furthermore the right emphasis on such planning, including the behaviour of the character, must also be realised.

The storyboard should serve as a blueprint for any film project and as the first visual impression of the film. It is at this stage that the major decisions are taken as far as the film's content is concerned. It is generally accepted that no production should proceed until a satisfactory storyboard is achieved and most of the creative and technical problems which may arise during the film's production have been considered.

There is no strict rule as to how many sketches are required for a film. It depends on the type, character and content of the project. A rough guideline is approximately 100 storyboard sketches for each minute of film. If, however, a film is technically complex, the number of sketches could double. For a TV commercial, more sketches are produced as a rule because there are usually more scene changes and more action than in longer films.

The basic unit of time in animation

The basis of timing in animation is the fixed projection speed of 24 frames per second (fps) for film and video. While other projection speeds have been used in the past the standard projection rate for film of all formats—16mm, 35mm and 70mm remains 24 fps. On television and video this becomes 25 frames per second (PAL) or 30 fps (NTSC), but the difference is usually imperceptible.



The thing to remember is that if an action on the screen takes one second it covers 24 frames of film, and if it takes half a second it covers 12 frames and so on.

24 frames of film go through the projector every second (25 on television). This fixed number of frames provides the basis on which all actions are planned and timed by the director.

For single frame animation, where one drawing is done for each frame, a second of action needs 24 drawings. If the same action is animated on double frames, where each drawing is photographed twice in succession, 12 drawings are necessary out the number of frames and hence the speed of the action would be the same in both cases.

Whatever the mood or pace of the action that appears on the screen, whether it be a frantic chase or a romantic love scene, all timing calculations must be based on the fact that the projector continues to hammer away at its constant projection rate. That is—24 fps for film and either 25 fps or 30 fps for television and video depending on format. The unit of time within which an animator works is, therefore, $1/24$ sec, $1/25$ sec or $1/30$ sec and an important part of the skill, which the animator has to learn is what this specific timing ‘feels’ like on screen. With practice the animator also learns what multiples of this unit look like—3 frames, 8 frames, 12 frames and so on.

Animation and properties of matter

The basic question which an animator is continually asking himself is: ‘What will happen to this object when a force acts upon it?’ And the success of his animation largely depends on how well he answers this question.

All objects in nature have their own weight, construction and degree of flexibility, and therefore each behaves in its own individual way when a force acts upon it. This behavior, a combination of position and timing, is the basis of animation. Animation consists of drawings, which have neither weight nor do they have any forces acting on them. In certain types of limited or abstract animation, the drawings can be treated as moving patterns. However, in order to give meaning to movement, the animator must consider Newton's laws of motion which contain all the information necessary to move characters and objects around. There are many aspects of his theories which are important in this book. However, it is not necessary to know the laws of motion in their verbal form, but in the way which is familiar to everyone, that is by watching things move. For instance, everyone knows that things do not start moving suddenly from rest—even a cannonball has to accelerate to its maximum speed when fired. Nor do things suddenly stop dead—a car hitting a wall of concrete carries on moving after the first impact, during which time it folds itself rapidly up into a wreck.

It is not the exaggeration of the weight of the object which is at the centre of animation, but the exaggeration of the tendency of the weight—any weight—to move in a certain way.

The timing of a scene for animation has two aspects:

1. The timing of inanimate objects
2. The timing of the movement of a living character

With inanimate objects the problems are straightforward dynamics. ‘How long does a door take to slam?’, ‘How quickly does a cloud drift across the sky?’, ‘How long does it take a steamroller, running out of control downhill, to go through a brick wall?’.

With living characters the same kind of problems occur because a character is a piece of flesh which has to be moved around by the action of forces on it. In addition, however, time must be allowed for the mental operation of the character, if he is to come alive on the screen. He must appear to be thinking his way through his actions, making decisions and finally moving his body around under the influence of his own will power and muscle.

Animation consists of sequences of weightless drawings. The success of animation on the screen depends largely on how well these drawings give the impression of reacting in an exaggerated way when weight and forces are made to act upon them.

Movement and caricature

The movement of most everyday objects around us is caused by the effect of forces acting upon matter.

The movement of objects becomes so familiar to us that, subconsciously, these movements give us a great deal of information about the objects themselves and about the forces acting on them. This is true not only of inanimate objects, but also of living things—especially people.

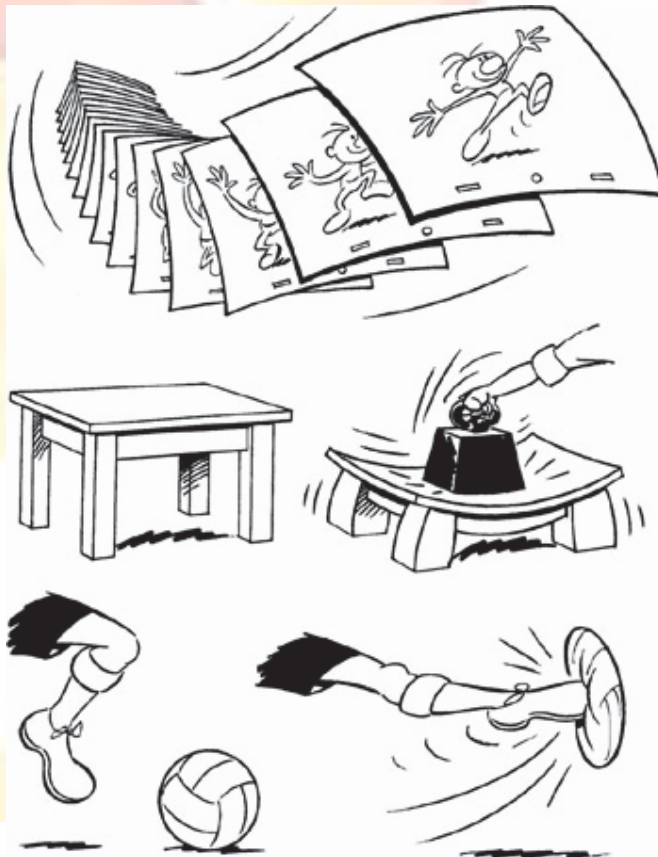
The animator's job is to synthesize movements and to apply just the right amount of creative exaggeration to make the movement look natural within the cartoon medium.

Cartoon film is a medium of caricature. The character of each subject and the movement it expresses are exaggerated. The subjects can be considered as caricatured matter acted upon by caricatured forces.

Cartoon film can also be a dramatic medium. This particular quality can be achieved, among other means, by speeded up action and highly exaggerated timing. The difference between an action containing caricature, or humour or drama, may be very subtle. Eventually, with enough experience in animation timing, it becomes possible to emphasise the difference.

Caricatured matter has the same properties as natural matter, only more so. To understand how cartoon matter behaves it is necessary to look more closely into the way matter behaves naturally.

Cartoon is a medium of caricature—naturalistic action looks weak in animation. Look at what actually happens, simplify down to the essentials of the movement and exaggerate these to the extreme.



Cause and effect

There is a train of cause and effect which runs through an object when it is acted upon by a force. This is the result of the transmission of the force through a more or less flexible medium (ie caricatured matter). This is one aspect of good movement in animation.

An animator must understand the mechanics of the natural movement of an object and then keep this knowledge in the back of his mind whilst he concentrates on the real business of animation. This is the creation of mood and conveying the right feeling by the way an action is done.

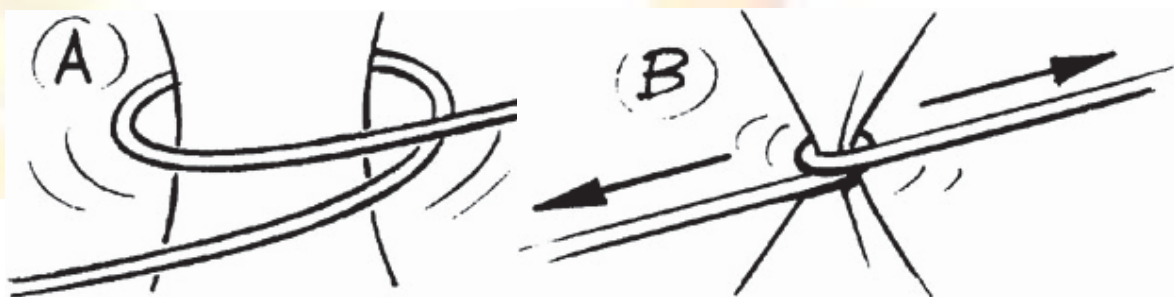


Examples of cause and effect:

Figs A and B

A rope wrapped around *anything* and pulled tight has the tendency shown. How far the reaction goes depends on:

- i. the strength of the forces pulling the rope.
- ii. the flexibility or rigidity of the object being squeezed. Exaggerate the tendency.

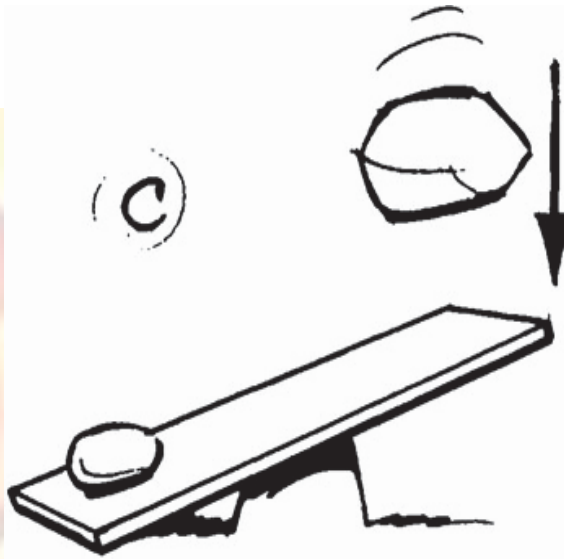


A A rope wrapped round something.

B The ends of the rope are pulled tight.

Fig. C

On the seesaw, the end of the plank with the smaller stone tends to stay where it is to begin with because of its inertia, so bending plank **D**. A moment later it starts to accelerate and the plank springs back with the opposite curvature causing the stone to whizz out of the screen, **E**.



C A stone on one end of a seesaw.

D and **E** A bigger stone drops onto the other end.

Newton's laws of motion

Every object or character has weight and moves only when a force is applied to it. This is Newton's first law of motion. An object at rest tends to remain at rest until a force moves it and once it is moving it tends to keep moving in a straight line until another force stops it.

The heavier an object is, or strictly speaking, the greater its mass, the more force is required to change its motion. A heavy body has more inertia and more momentum than a light one.

A heavy object at rest, such as a cannonball, needs a lot of force to move it (following Fig. A). When fired from a cannon, the force of the charge acts on the cannonball only whilst it is in the gun barrel. Since the force of

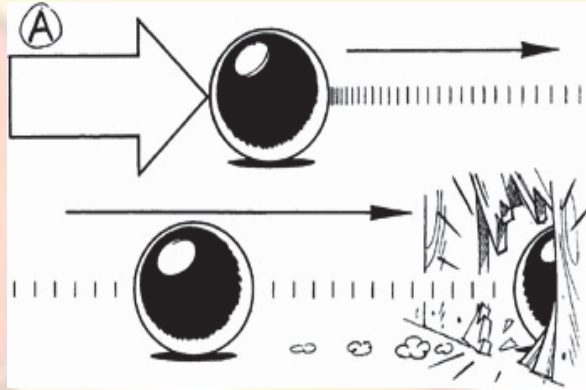


the explosive charge is very large indeed, this is sufficient to accelerate the cannonball to a considerable speed. A smaller force acting for a short time, say a strong kick, may have no effect on the cannonball at all. In fact it is more likely to damage the kicker's toe. However, persistent force, even if not very strong, would gradually start the cannonball rolling and it would eventually be travelling fairly quickly.

A A cannon ball needs a lot of force to start it moving. Once moving, it takes a lot of stopping.

Once the cannonball is moving, it tends to keep moving at the same speed and requires some force to stop it. If it meets an obstacle it may, depending on its speed, crash straight through it. If it is

rolling on a rough surface it comes to rest fairly soon, but if rolling on a smooth flat surface, friction takes quite a long time to bring it to rest.



When dealing with very heavy objects, therefore, the director must allow plenty of time to start, stop or change their movements, in order to make their weight look convincing. The animator, for his part, must see that plenty of force is applied to the cannonball to make it start, stop or change direction.

Light objects have much less resistance to change of movement and so behave very differently when forces act on them. A toy balloon (Fig. B) needs much less time to start it moving. The flick of a finger is enough to make it accelerate quickly away. When moving, it has little momentum and the friction of the air quickly slows it up, so it does not travel very far.

B A balloon needs only a small force to move it, but air resistance quickly brings it to rest. In both these examples a circle is being animated. The timing of its movements can make it look heavy or light on the screen.



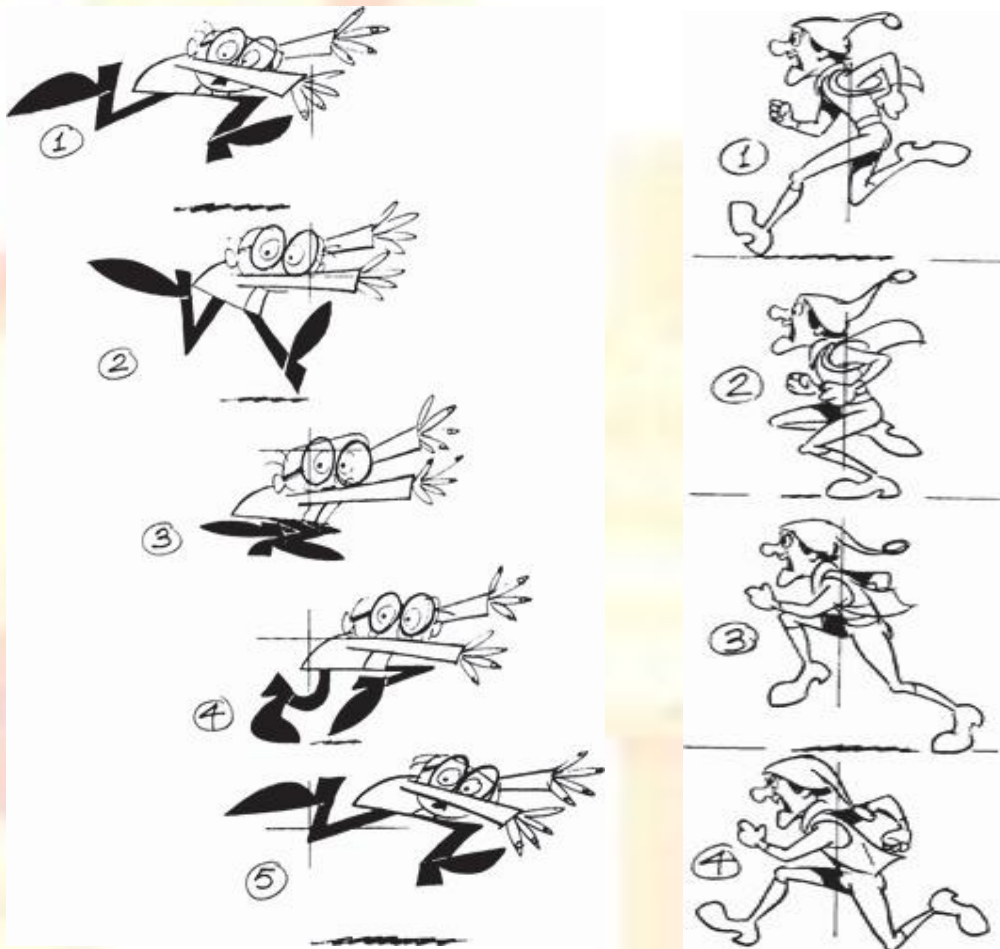
The way an object behaves on the screen, and

the effect of weight that it gives, depends entirely on the spacing of the animation drawings and not on the drawing itself. It does not matter how beautifully drawn the cannonball is in the static sense, it does not look like a cannonball if it does not behave like one and the same applies to the balloon, and indeed to any other object or character.

Fast run cycles

An eight frame run cycle—that is four frames to each step—gives a fast and vigorous dash. At this speed the successive leg positions are quite widely separated and may need drybrush or speed lines to make the movement flow. Drawing 5 shows the same position as drawing 1 but with opposite arms and feet. Similarly drawings 6 and 2, 7 and 3, and 8 and 4 show the same positions. These alternate positions should be varied slightly in each case, to avoid the rather mechanical effect of the same positions occurring every four frames.

A twelve frame cycle gives a less frantic run, but if the cycle is more than sixteen frames the movement tends to lose its dash and appear too leisurely.



The body normally leans forward in the direction of movement, although for comic effect a backward lean can sometimes work. If a faster run than an eight frame repeat is needed, then perhaps several foot positions can be given on each drawing, to fill up the gaps in the movement, or possibly the legs can become a complete blur treated entirely in dry-brush.

In the first example, drawing 4 is equivalent to the 'step' position in a walk, with the maximum forward and backward leg and arm movement. In a run it is also the point at which the centre of gravity of the body is farthest from the ground, that is,

in mid-stride. In drawing 1 the weight is returning to its lowest point, which is in drawing 2. In drawing 3 the body starts to rise again as the thrust of the back foot gives the forward impetus for the next stride.

These are both examples of eight frame run cycles. This means four drawings to each step. Drawings 1 and 5 show the same leg and arm positions but with opposite feet, and so do 2 and 6, 3 and 7, 4 and 8. In such a short cycle these positions should be varied slightly to avoid a mechanical effect.

Timing and music

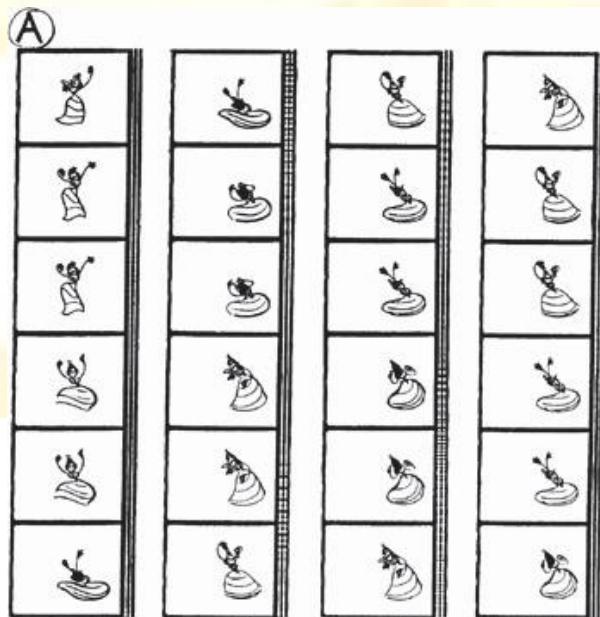
Ever since the very first animated productions, Disney's *Steamboat Mickey* and Fischinger's abstract film *Brahm's Hungarian Dances*, it was clear that there is a strong relationship between animation and music. This relationship can be explained on two accounts. First, both elements have a basic mathematical foundation and move forward at a determined speed. Second, since animation is created manually frame by frame, it can be fitted to music in a very exact manner. It is further able to capture its rhythm, its mood and hit the beat right to the frame. Most animation makes good use of this advantage.

In general principle it is more difficult to follow the rhythm of a musical composition with its mood than its beat. The latter aspect of the music is easily measured, since beats are fitted into bar units of defined time length and are interpreted in time units.

Bars can contain various numbers of beats and these must be measured to the film frame. Having done this, it is comparatively easy to fit the animation to the speed of the beat and find the right type of movement to follow the music, whether it is a slow waltz of 36 frames, or 4 frames for rock music. A beat can be emphasised by synchronisation of the feet but it works better if the whole body is used. In quick beats of 3,4 or 6 frames it is possible to follow every second beat without losing the rhythm. It is always better to work to specially prepared music if this can be afforded.

A This eight frame cycle, animated on double frames, of a whirling Spanish dancer is fitted closely to strong flamenco music. The figure fully conveys the character of the music with all its functional simplicity.

B Single and double frame animation are alternated to fit the beats of the sound of a Spanish guitar. It is essential for the movement to follow the musical lead of a specially



prerecorded music track, for accurate synchronisation.



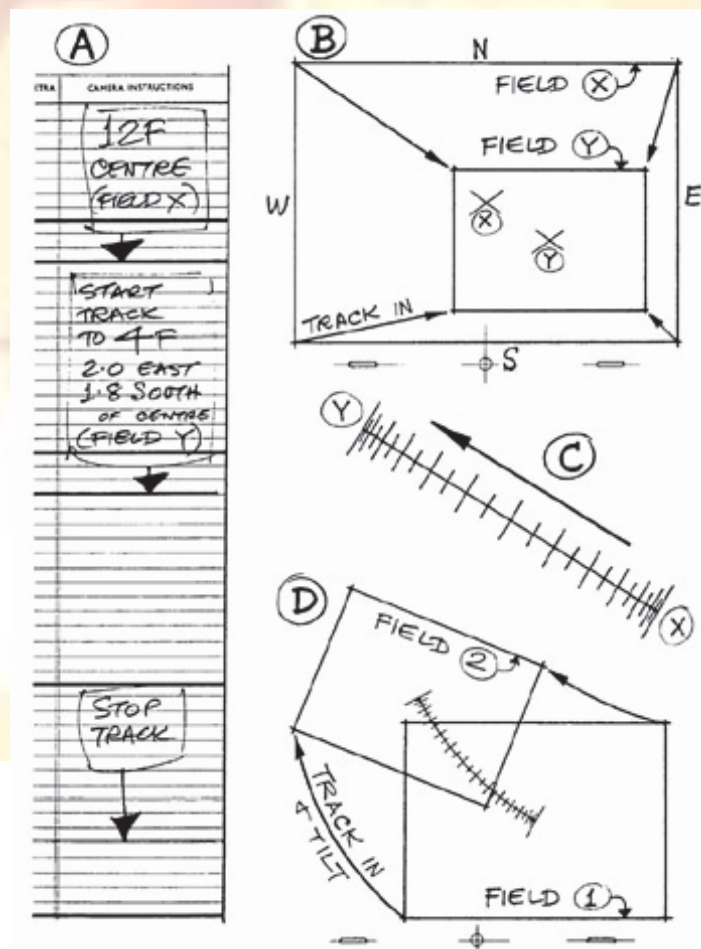
Camera movements

Tracks are used to move into a closer field or pull back to a more distant one. They are done by moving the camera frame by frame, up or down its vertical pillar, above the animation drawings. Usually the field centre also moves during a track and as the camera travels on a fixed axis this movement in a north south, east west direction is done by moving the table.

Tracks and table moves are worked out in terms of general timing—track lengths and the action which the various fields must include—by the director on bar sheets before production begins.

When the animator finalises the action of the scene in detail, he converts the director's timing into specific instructions to the cameraman. He writes down the field sizes and marks the frames where camera movements start and stop in the 'camera instructions' column on the exposure chart (Fig. A). He also provides a drawn field key with field centres marked (Fig. B).

It then becomes the cameraman's responsibility to achieve the required effect smoothly and accurately on the screen. Briefly, the procedure is as follows: in Fig. B the track is made from field X to field Y so the screen centre moves towards the south-east. This means that under the camera, the table must move north-west. Fig. C is an enlargement of this table move, showing how the cameraman divides the line to achieve a smooth movement from X to Y. At the same time he measures the distance the camera travels on its column during the track, and divides this



in exactly the same way as Fig. C, so that camera and table top move smoothly together. Fig. D is a similar track and table move which includes a tilt. This would also be done as a table move.

Tracks and table moves are usually animated on single frames.

A The director's timing of tracks is finalised by the animator in the 'camera instructions' column on the exposure sheet.

B The accompanying field key.

C Enlargement of field centres from Fig. B, inverted for use as table move.

D Another example of a track including a table tilt.

