GP QB Unit 2

1. **Write a short note on SINE Rule.**
2. **Write a short note on COSINE Rule.**

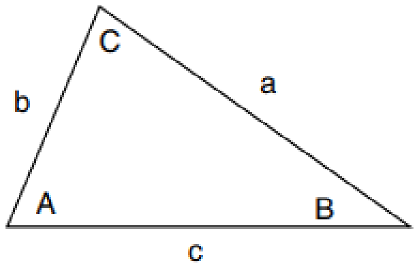


Fig. 4.3. An arbitrary triangle.

4.5 The Sine Rule

The sine rule relates angles and side lengths for a triangle. Figure 4.3 shows a triangle labelled such that side a is opposite angle A, side b is opposite angle B, etc.

The sine rule states

$$\frac{a}{\sin A} = \frac{b}{\sin B} = \frac{c}{\sin C}$$

4.6 The Cosine Rule

The cosine rule expresses the $\sin^2(\beta) + \cos^2(\beta) = 1$ relationship for the arbitrary triangle shown in Figure 4.3. In fact, there are three versions:

$$a^2 = b^2 + c^2 - 2bc\ \cos(A)$$
$$b^2 = c^2 + a^2 - 2ca\ \cos(B)$$
$$c^2 = a^2 + b^2 - 2ab\ \cos(C)$$

Three further relationships also hold:

$$a = b\ \cos(C) + c\ \cos(B)$$
$$b = c\ \cos(A) + a\ \cos(C)$$
$$c = a\ \cos(B) + b\ \cos(A)$$

3. Explain Bernstein Polynomials. [incomplete]

Bézier curves employ *Bernstein polynomials*, which were described by S. Bernstein in 1912. They are expressed as follows:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{9.5}$$

where $\binom{n}{i}$ is shorthand for the number of selections of i different items from n distinguishable items when the order of selection is ignored, and equals

$$\frac{n!}{(n-i)!i!} \tag{9.6}$$

where, for example, 3! (factorial 3) is shorthand for $3 \times 2 \times 1$.

When (9.6) is evaluated for different values of i and n, we discover the pattern of numbers.

This pattern of numbers is known as Pascal's triangle.

The pattern represents the coefficients found in binomial expansions.

4. Explain B-Splines and its types. OR

Write a short note on B-splines.

- B-splines, like B´ezier curves, use polynomials to generate a curve segment.
- But, unlike B´ezier curves, B-splines employ a series of control points that determine the curve's local geometry. This feature ensures that only a small portion of the curve is changed when a control point is moved.
- There are two types of B-splines: rational and non-rational splines, which divide into two further categories: uniform and non-uniform.
- Rational B-splines are formed from the ratio of two polynomials such as

$$x(t) = \frac{X(t)}{W(t)},\ y(t) = \frac{Y(t)}{W(t)},\ z(t) = \frac{Z(t)}{W(t)},$$

- Uniform B-Splines
    - A B-spline is constructed from a string of curve segments whose geometry
    - is determined by a group of local control points. These curves are known
    - as piecewise polynomials. A curve segment does not have to pass through a
    - control point, although this may be desirable at the two end-points
- Continuity
    - Constructing curves from several segments can only succeed if the slopes of
    - the abutting curves match. As we are dealing with curves whose slopes are
    - changing everywhere, it is necessary to ensure that even the rate of change of
    - slopes is matched at the join. This aspect of curve design is called geometric
    - continuity and is determined by the continuity properties of the basis function.
- Non-Uniform B-Splines
    - Uniform B-splines are constructed from curve segments where the parameter
    - spacing is at equal intervals. Non-uniform B-splines, with the support of a
    - knot vector, provide extra shape control and the possibility of drawing periodic
    - shapes

- Non-Uniform Rational B-Splines
    - Non-uniform rational B-splines (NURBS) combine the advantages of nonuniform
    - B-splines and rational polynomials: they support periodic shapes such
    - as circles, and they accurately describe curves associated with the conic sections.
    - They also play a very important role in describing geometry used in the
    - modelling of computer animation characters.

5. What are Bezier Curves? How Bernstein Polynomials are used to interpolate the Bezier Curves?

- Bézier curves are widely used in computer graphics to model smooth curves.
- As the curve is completely contained in the convex hull of its control points, the points can be graphically displayed and used to manipulate the curve intuitively.
- Affine transformations such as translation and rotation can be applied on the curve by applying the respective transform on the control points of the curve.

6. Explain the types of Bezier Curve.

9.3.1 Bernstein Polynomials

Bézier curves employ *Bernstein polynomials*, which were described by S. Bernstein in 1912. They are expressed as follows:

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-i} \tag{9.5}$$

where $\binom{n}{i}$ is shorthand for the number of selections of i different items from n distinguishable items when the order of selection is ignored, and equals

$$\frac{n!}{(n-i)!i!} \tag{9.6}$$

•

9.3.2 Quadratic Bézier Curves

Quadratic Bézier curves are formed by using Bernstein polynomials to interpolate between the x-, y- and z-coordinates associated with the start- and end-points forming the curve. For example, we can draw a 2D quadratic Bézier curve between (1, 1) and (4, 3) using the following equations:

$$\begin{aligned} x &= 1(1-t)^2 + x_c 2t(1-t) + 4t^2 \\ y &= 1(1-t)^2 + y_c 2t(1-t) + 3t^2 \end{aligned} \tag{9.11}$$

•

9.3.3 Cubic Bernstein Polynomials

One of the problems with quadratic curves is that they are so simple. If we wanted to construct a complex curve with several peaks and valleys, we would have to join together a large number of such curves. A *cubic curve*, on the other hand, naturally supports one peak and one valley, which simplifies the construction of more complex curves.

When $n = 3$ in (9.7), we obtain the following terms:

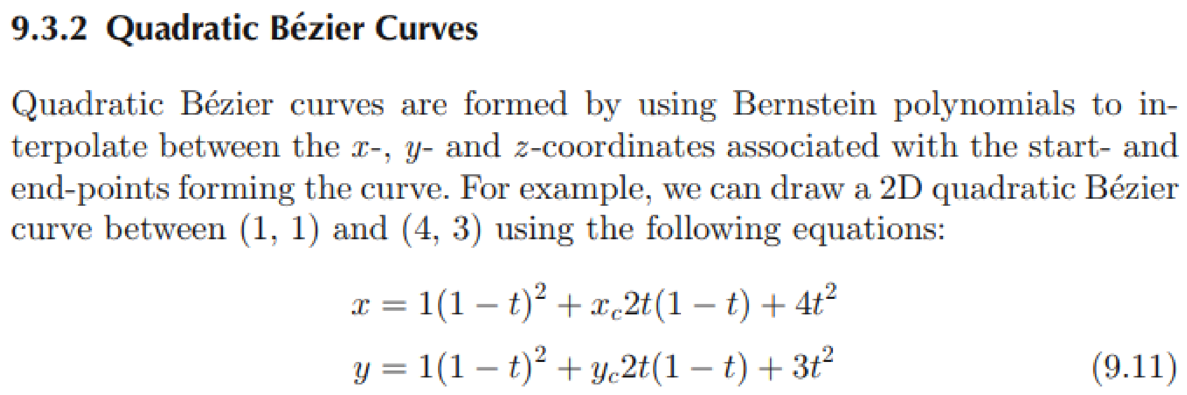
$$((1-t)\ t)^3 = (1-t)^3 + 3t(1-t)^2 + 3t^2(1-t) + t^3 \tag{9.12}$$

which can be used as a cubic interpolant, as

$$V = V_1(1-t)^3 + V_{c1}3t(1-t)^2 + V_{c2}3t^2(1-t) + V_2 t^3 \tag{9.13}$$

•

7. **What is Bezier curve? Explain quadratic bezier curve.**

- Bézier curves are widely used in computer graphics to model smooth curves.
- As the curve is completely contained in the convex hull of its control points, the points can be graphically displayed and used to manipulate the curve intuitively.
- Affine transformations such as translation and rotation can be applied on the curve by applying the respective transform on the control points of the curve.
- 

  9.3.2 Quadratic Bézier Curves

  Quadratic Bézier curves are formed by using Bernstein polynomials to interpolate between the x-, y- and z-coordinates associated with the start- and end-points forming the curve. For example, we can draw a 2D quadratic Bézier curve between (1, 1) and (4, 3) using the following equations:

  $$x = 1(1-t)^2 + x_c 2t(1-t) + 4t^2$$
  $$y = 1(1-t)^2 + y_c 2t(1-t) + 3t^2 \qquad (9.11)$$

8. Explain the concept of swap Chain and page flipping.

- Swap Chain
  - A swap chain is a collection of buffers that are used for displaying frames to the user. Each time an application presents a new frame for display, the first buffer in the swap chain takes the place of the displayed buffer. This process is called swapping or flipping.
  - In every swap chain there are at least two buffers. The first framebuffer, the screenbuffer, is the buffer that is rendered to the output of the video card. The remaining buffers are known as backbuffers.
  - Each time a new frame is displayed, the first backbuffer in the swap chain takes the place of the screenbuffer, this is called presentation or swapping.
  - A graphics adapter holds a pointer to a surface that represents the image being displayed on the monitor, called a front buffer. As the monitor is refreshed, the graphics card sends the contents of the front buffer to the monitor to be displayed.

- Page Flipping
  - In the page-flip method, instead of copying the data, both buffers are capable of being displayed (both are in VRAM).
  - At any one time, one buffer is actively being displayed by the monitor, while the other, background buffer is being drawn. When the background buffer is complete, the roles of the two are switched.
  - The page-flip is typically accomplished by modifying the value of a pointer to the beginning of the display data in the video memory.
  - The page-flip is much faster than copying the data and can guarantee that tearing will not be seen.
  - The currently active and visible buffer is called the front buffer, while the background page is called the "back buffer".

9. Write a short note on depth buffering.

- One of the simplest and commonly used image space approach to eliminate hidden surfaces is the Z-buffer or depth buffer algorithm.
- This algorithm compares surface depths at each pixel position on the projection plane. The surface depth is measured from the view plane along the z-axis of a viewing system.
- When object description is converted to projection co-ordinates (x, y, z), each pixel position on the view plane is specified by x and y coordinate, and z-value gives the depth information. Thus, object depths can be compared by comparing the z- values.
- The Z-buffer algorithm is usually implemented in the normalized coordinates, so that z-values range from 0 at the back-clipping plane to 1 at the front clipping plane.
- The implementation requites another buffer memory called Z-buffer along with the frame buffer memory required for raster display devices.
- A Z-buffer is used to store depth values for each (x, y) position as surfaces are processed, and the frame buffer stores the intensity values for each position.
- At the beginning Z-buffer is initialized to zero, representing the z-value at the back-clipping plane, and the frame buffer is initialized to the background colour.

10. Explain the concept of multi-sampling theory. Describe how multi- sampling is done in Direct3D.

- Because the pixels on a monitor are not infinitely small, an arbitrary line cannot be represented perfectly on the computer monitor.
- a “stair-step” (aliasing) effect can occur when approximating a line by a matrix of pixels.
- Similar aliasing effects occur with the edges of triangles.
- we observe aliasing (the stair-step effect when trying to represent a line by a matrix of pixels).
- an antialiased line, generates the final color of a pixel by sampling and using its neighboring pixels; this results in a smoother image and dilutes the stair-step effect.
- Shrinking the pixel sizes by increasing the monitor resolution can alleviate the problem significantly to where the stair-step effect goes largely unnoticed.
- Multisampling in Direct3D
  - DXGI_SAMPLE_DESC structure. This structure has two members and is defined as follows:
  - typedef struct DXGI_SAMPLE_DESC {
    UINT Count;
    UINT Quality;
    } DXGI_SAMPLE_DESC, *LPDXGI_SAMPLE_DESC;
  - The Count member specifies the number of samples to take per pixel
  - Quality member is used to specify the desired quality level (what “quality level” means can vary across hardware manufacturers).
  - Use the following method to query the number of quality levels for a given texture format and sample count:
  - HRESULT ID3D11Device::CheckMultisampleQualityLevels(DXGI_FORMAT Format, UINT SampleCount, UINT*pNumQualityLevels);
  - This method returns zero if the format and sample count combination is not supported by the device. Otherwise, the number of quality levels for the given combination will be returned through the pNumQualityLevels parameter.
  - Valid quality levels for a texture format and sample count combination range from zero to pNumQualityLevels −1.
  - The maximum number of samples that can be taken per pixel is defined by:
    #define D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT (32)

    - However, a sample count of 4 or 8 is common in order to keep the performance and memory cost of multisampling reasonable.

11. What is COM? Explain the texture and resources format in DirectX.

- COM
  - Component Object Model (COM) is the technology that allows DirectX to be programming language independent and have backwards compatibility.
  - Most of the details of COM are hidden to us when programming DirectX with C++. The only thing that we must know is that we
    - obtain pointers to COM interfaces through special functions or by the methods of another COM interface–we do not create a COM
    - interface with the C++ new keyword.
  - In addition, when we are done with an interface, we call its Release method (all COM interfaces inherit functionality from the IUnknown COM interface, which provides the Release method) rather than delete it
  - COM objects perform their own memory management.
- Textures and Data Resource Formats
  - A 2D texture is a matrix of data elements. One use for 2D textures is to store 2D image data, where each element in the texture stores the color of a pixel.
  - However, this is not the only usage; for example, in an advanced technique called normal mapping, each element in the texture stores a 3D vector instead of a color.
  - Therefore, although it is common to think of textures as storing image data, they are really more general purpose than that. A 1D texture is like a 1D array of data elements, and a 3D texture is like a 3D array of data elements.
  - In addition, a texture cannot store arbitrary kinds of data; it can only store certain kinds of data formats, which are described by the DXGI_FORMAT enumerated type

13. State and define different trigonometry ratio and inverse of trigonometric ratio.

- **Trigonometric Ratios**

Today, the trigonometric ratios are commonly known by the abbreviations sin, cos, tan, cosec, sec and cot. Figure 4.1 shows a right-angled triangle where the trigonometric ratios are given by

$$\sin(\beta) = \frac{opposite}{hypotenuse} \qquad \cos(\beta) = \frac{adjacent}{hypotenuse} \qquad \tan(\beta) = \frac{opposite}{adjacent}$$

$$\text{cosec}(\beta) = \frac{1}{\sin(\beta)} \qquad \sec(\beta) = \frac{1}{\cos(\beta)} \qquad \cot(\beta) = \frac{1}{\tan(\beta)}$$

The sin and cos functions have limits ± 1, whereas tan has limits $\pm\infty$. The signs of the functions in the four quadrants are

$$\begin{array}{c|c} + & + \\ \hline - & - \end{array} \qquad \begin{array}{c|c} - & + \\ \hline - & + \end{array} \qquad \begin{array}{c|c} - & + \\ \hline + & - \end{array}$$

$$\text{sin} \qquad\quad \text{cos} \qquad\quad \text{tan}$$

-

4.3 Inverse Trigonometric Ratios

As every angle has its associated ratio, functions are required to convert one into the other. The sin, cos and tan functions convert angles into ratios, and the inverse functions $\sin^{-1}, \cos^{-1}$ and $\tan^{-1}$ convert ratios into angles. For example, $\sin(45°) = 0.707$, therefore $\sin^{-1}(0.707) = 45°$. Although the sin and cos functions are *cyclic* functions (i.e. they repeat indefinitely) the inverse functions return angles over a specific period.

-

14. What is interpolation? explain linear interpolation.

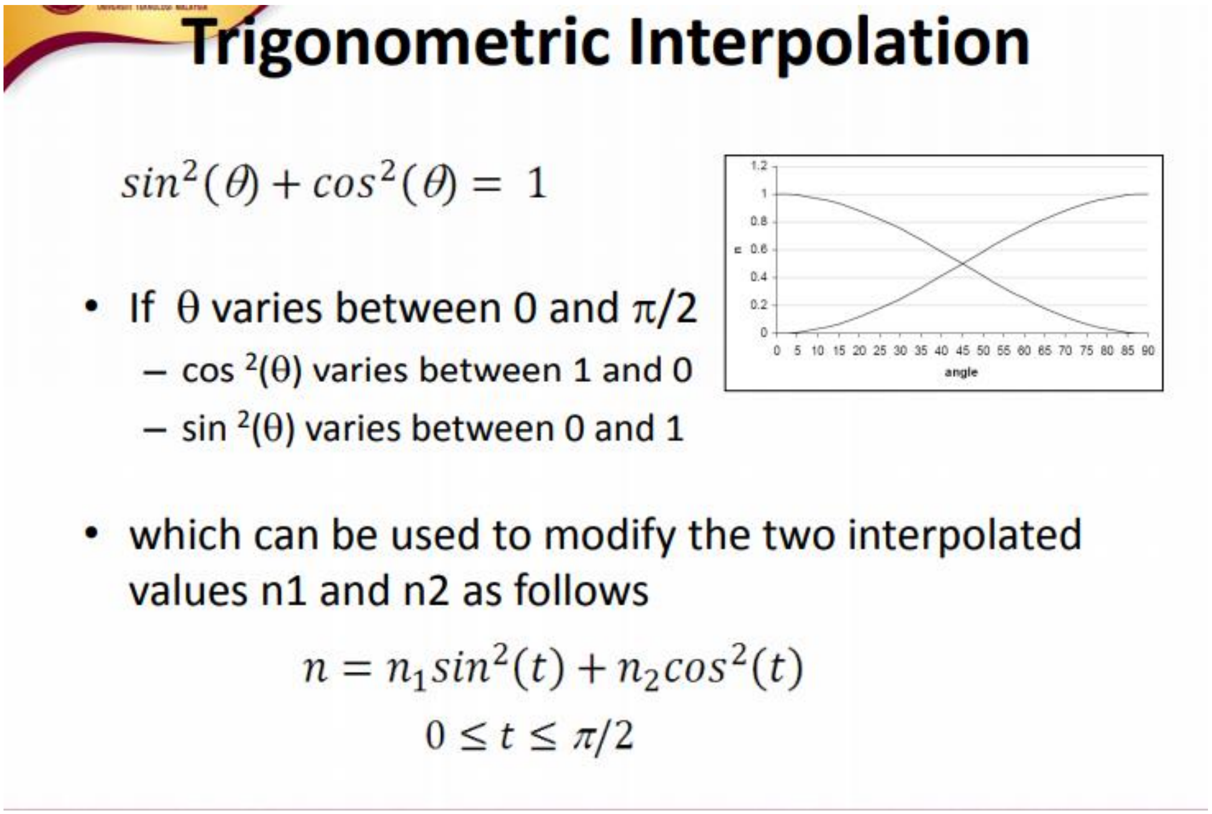
- In the context of computer animation, interpolation is inbetweening, or filling in frames between the key frames.
- It typically calculates the in between frames through use of (usually) piecewise polynomial interpolation to draw images semi-automatically.
- Basically, an interpolant is a way of changing one number into another.
- The real function of an interpolant is to change one number into another in n equal steps
- Linear Interpolant
    - A linear interpolant generates equal spacing between the interpolated values for equal changes in the interpolating parameter.
    - Given two numbers n1 and n2, which represent the start and final values of the interpolant, we require an interpolated value controlled by a parameter t that varies between 0 and 1.
    - When t = 0, the result is n1, and when t = 1, the result is n2.
    - A solution to this problem is given by

$$n = n1 + t(n2 - n1)$$

15. Explain the following interpolation

a. trigonometric interpolation

b. cubic interpolation

- 

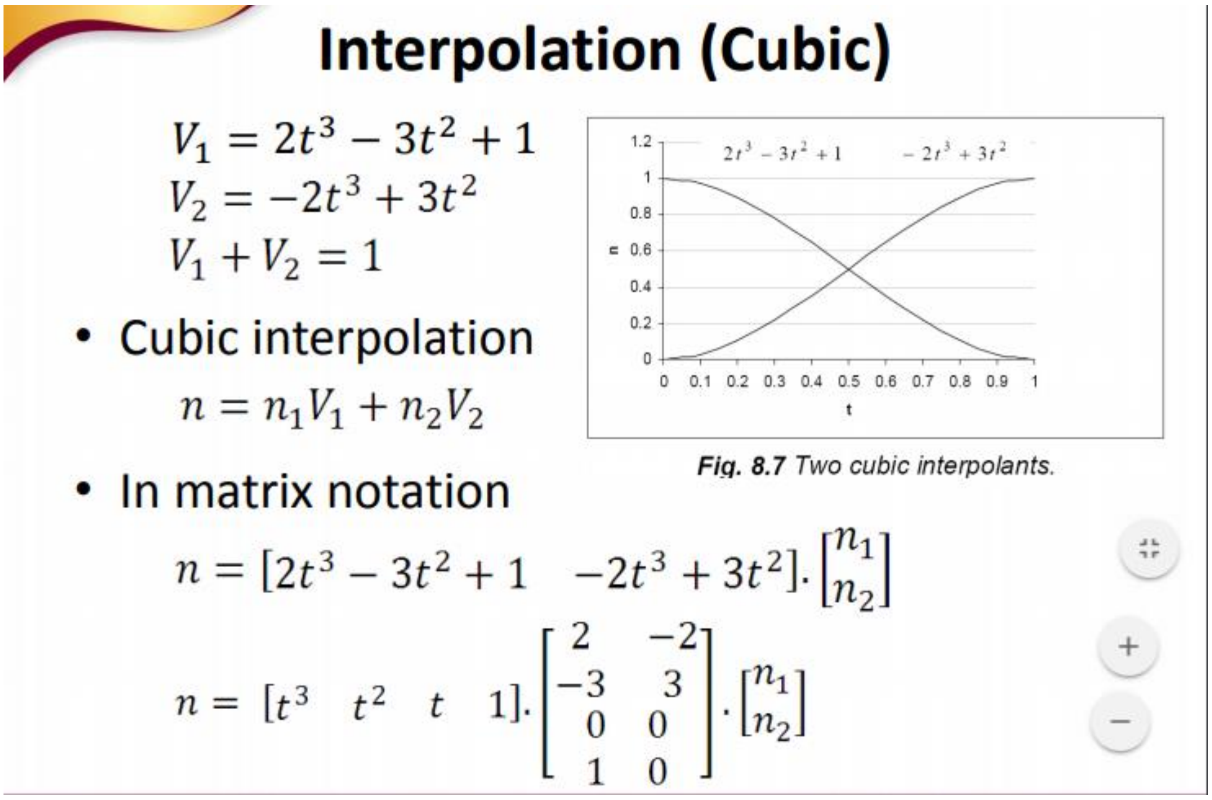
Trigonometric Interpolation

$$\sin^2(\theta) + \cos^2(\theta) = 1$$

- If θ varies between 0 and $\pi/2$
  - $\cos^2(\theta)$ varies between 1 and 0
  - $\sin^2(\theta)$ varies between 0 and 1
- which can be used to modify the two interpolated values n1 and n2 as follows

$$n = n_1 \sin^2(t) + n_2 \cos^2(t)$$
$$0 \le t \le \pi/2$$

-

- 

Interpolation (Cubic)

$$V_1 = 2t^3 - 3t^2 + 1$$
$$V_2 = -2t^3 + 3t^2$$
$$V_1 + V_2 = 1$$

- Cubic interpolation

$$n = n_1 V_1 + n_2 V_2$$

- In matrix notation

$$n = [2t^3 - 3t^2 + 1 \quad -2t^3 + 3t^2] . \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

$$n = [t^3 \quad t^2 \quad t \quad 1] . \begin{bmatrix} 2 & -2 \\ -3 & 3 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} . \begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

Fig. 8.7 Two cubic interpolants.

-

17. Explain the ways to examine to test whether the point is inside, outside or touching a triangle.

- Area of a Triangle
  - The area of the triangle is given by:

    $$A = \frac{1}{2}[x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)]$$

  - If the area of triangle (P1, P2, Pt) is positive, Pt must be to the left of the line (P1, P2).
  - If the area of triangle (P2, P3, Pt) is positive, Pt must be to the left of the line (P2, P3).
  - If the area of triangle (P3, P1, Pt) is positive, Pt must be to the left of the line (P3, P1).
  - If all the above tests are positive, Pt is inside the triangle.
  - Furthermore, if one area is zero and the other areas are positive, the point is on the boundary, and if two areas are zero and the other positive, the point is on a vertex
- Hessian Normal Form
  - We can determine whether a point is inside, touching or outside a triangle by representing the triangle's edges in the Hessian normal form, and testing which partition the point is located in.
  - If we arrange that the normal vectors are pointing towards the inside of the triangle, any point inside the triangle will create a positive result when tested against the edge equation
  - We are only interested in the sign of the left-hand expressions, which can be tested for any arbitrary point (x, y).
  - If they are all positive, the point is inside the triangle.
  - If one expression is negative, the point is outside.
  - If one expression is zero, the point is on an edge, and if two expressions are zero, the point is on a vertex.

18. Explain lambert's cosine law.

- Lambert's law states that the intensity of illumination on a diffuse surface is proportional to the cosine of the angle between the surface normal vector and the light source direction.
- An important consequence of Lambert's cosine law is that when an area element on the surface is viewed from any angle, it has the same radiance.

19. Explain the following lightning

a. Diffuse lighting

b. Ambient lighting

c. Specular lighting

- Diffuse Lighting
  - When light strikes a point on such a surface, the light rays scatter in various random directions; this is called a diffuse reflection.
  - In our approximation for modelling this kind of light/surface interaction, we stipulate that the light scatters equally in all directions above the surface; consequently, the reflected light will reach the eye no matter the viewpoint (eye position). Therefore, we do not need to take the viewpoint into consideration (i.e., the diffuse lighting calculation is viewpoint
  - independent), and the color of a point on the surface will always look the same no matter the viewpoint.
- Ambient lighting
  - Much of the light we see in the real world is indirect. For example, a hallway connected to a room might not be in the direct line of sight with a light source in the room, but the light bounces off the walls in the room and some of it may make it into the hallway, thereby lightening it up a bit.
  - As a second example, suppose we are sitting in a room with a teapot on a desk and there is one light source in the room. Only one side of the teapot is in the direct line of sight of the light source; nevertheless, the backside of the teapot would not be pitch black.
  - This is because some light scatters off the walls or other objects in the room and eventually strikes the backside of the teapot
  - This is called Ambient Lighting
- Specular lighting
  - When light strikes such a surface, the light rays reflect sharply in a general direction through a cone of reflectance; this is called a specular reflection.
  - In contrast to diffuse light, specular light might not travel into the eye because it reflects in a specific direction; the specular lighting calculation is viewpoint dependent.
  - This means that as the eye moves about the scene, the amount of specular light it receives will change.

20. State the difference between parallel light and spotlight.

- Parallel Lights
    - A parallel light (or directional light) approximates a light source that is very far away.
    - Consequently, we can approximate all incoming light rays as parallel to each other.
    - A parallel light source is defined by a vector, which specifies the direction the light rays travel. Because the light rays are parallel, they all use the same direction vector.
    - The light vector aims in the opposite direction the light rays travel.
    - A common example of a real directional light source is the sun
- Spot Lights
    - A spot light behaves exactly how it sounds, like a real spot light, and provides a very direct source of light.
    - One of the key benefits that you get when using a spot light is the directionally that you get from the light.
    - The spot light is emitted through a cone and you can control how wide the cone angle is which determines how much of the area is actually illuminated.
    - Objects closer to the spot light will be brighter, and depending on the how wide the cone is the light will either be softer or harder.
    - A good physical example of a spotlight is a flashlight.

21. Explain in brief magnification and minification.

- Magnification
    - The elements of a texture map should be thought of as discrete color samples from a continuous image; they should not be thought of as rectangles with areas.
    - So the question is: What happens if we have texture coordinates (u, v) that do not coincide with one of the texel points?
    - Suppose the player zooms in on a wall in the scene so that the wall covers the entire screen.
    - For the sake of example, suppose the monitor resolution is 1024 × 1024 and the wall's texture resolution is 256 × 256.
    - This illustrates texture magnification—we are trying to cover many pixels with a few texels.
    - In our example, between every texel point lies four pixels. Each pixel will be given a pair of unique texture coordinates when the vertex texture coordinates are interpolated across the triangle.
    - Thus there will be pixels with texture coordinates that do not coincide with one of the texel points.
    - Given the colors at the texels we can approximate the colors between texels using interpolation.

- Minification
    - Minification is the opposite of magnification. In minification, too many texels are being mapped to too few pixels.
    - For instance, consider the following situation where we have a wall with a 256 × 256 texture mapped over it. The eye, looking at the wall, keeps moving back so that the wall gets smaller and smaller until it only covers 64 × 64 pixels on screen.
    - So now we have 256 × 256 texels getting mapped to 64 × 64 screen pixels. In this situation, texture coordinates for pixels will still generally not coincide with any of the texels of the texture map, so constant and linear interpolation filters still apply to the minification case.
    - However, there is more that can be done with minification. Intuitively, a sort of average downsampling of the 256 × 256 texels should be taken to reduce it to 64 × 64.
    - The technique of mipmapping offers an efficient approximation for this at the expense of some extra memory.

22. Explain texture coordinates and state how to create and enable texture.

- Texture Coordinates
    - Texture coordinates define how an image (or portion of an image) gets mapped to a geometry.
    - A texture coordinate is associated with each vertex on the geometry, and it indicates what point within the texture image should be mapped to that vertex.
    - Texture coordinates are not stored with appearance, but on each geometry individually. This allows separate geometries to share an appearance with an image texture, yet display distinct portions of that image on each geometry.
    - u and v Texture Coordinates
      Each texture coordinate is, at a minimum, a (u,v) pair, which is the horizontal and vertical location in texture space, respectively.
    - Texture coordinates are used to define a triangle on the texture that gets mapped to the 3D triangle.
- CREATING AND ENABLING A TEXTURE
    - 1. Call D3DX11CreateTextureFromFile to create the ID3D11Texture2D object from an image file stored on disk.
    - 2. Call ID3D11Device::CreateShaderResourceView to create the corresponding shader resource view to the texture.

- Both of these steps can be done at once with the following D3DX function:
  HRESULT D3DX11CreateShaderResourceViewFromFile(
  ID3D11Device *pDevice,
  LPCTSTR pSrcFile,
  D3DX11_IMAGE_LOAD_INFO *pLoadInfo,
  ID3DX11ThreadPump *pPump,
  ID3D11ShaderResourceView **ppShaderResourceView,
  HRESULT *pHResult
  );

23. What is blending? State blend operation and blend factors.
28. What is Blending? Explain the Blending equation, Blend Operations, Blend Factors and Blend State.
29. Write a Note on Blending.

- Blending techniques allow us to blend (combine) the pixels that we are currently rasterizing (so-called source pixels) with the pixels that were previously rasterized to the back buffer (so-called destination pixels).
- This technique enables us, among other things, to render semi-transparent objects such as water and glass.
- THE BLENDING EQUATION
  - Let Csrc be the color output from the pixel shader for the ijth pixel we are currently rasterizing (source pixel), and let Cdst be the color of the ijth pixel currently on the back buffer (destination pixel).
  - Without blending, Csrc would overwrite and become the new color of the ijth back buffer pixel.
  - But, with blending Csrc and Cdst are blended together to get the new color C that will overwrite Cdst (i.e., the blended color C will be written to the ijth pixel of the back buffer).
  - Direct3D uses the following blending equation to blend the source and destination pixel colors:

$$\mathbf{C} = \mathbf{C}_{src} \otimes \mathbf{F}_{src} \boxplus \mathbf{C}_{dst} \otimes \mathbf{F}_{dst}$$

  -

9.2 BLEND OPERATIONS

The binary ⊞ operator used in the blending equation may be one of the following:

typedef enum D3D11_BLEND_OP
{

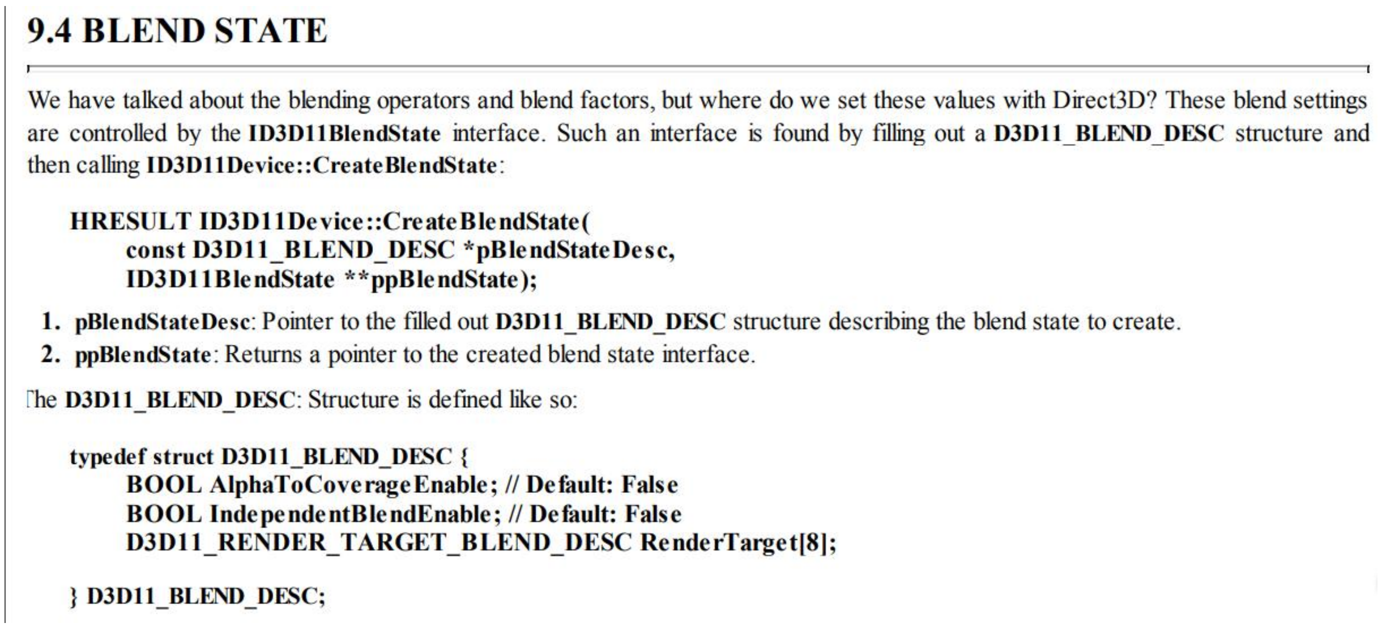
D3D11_BLEND_OP_ADD = 1,	$C = C_{src} \otimes F_{src} + C_{dst} \otimes F_{dst}$
D3D11_BLEND_OP_SUBTRACT = 2,	$C = C_{dst} \otimes F_{dst} - C_{src} \otimes F_{src}$
D3D11_BLEND_OP_REV_SUBTRACT = 3,	$C = C_{src} \otimes F_{src} - C_{dst} \otimes F_{dst}$
D3D11_BLEND_OP_MIN = 4,	$C = \min(C_{src}, C_{dst})$
D3D11_BLEND_OP_MAX = 5,	$C = \max(C_{src}, C_{dst})$

} D3D11_BLEND_OP;

9.3 BLEND FACTORS

By setting different combinations for the source and destination blend factors along with different blend operators, dozens of different blending effects may be achieved. We will illustrate some combinations in §9.5, but you will need to experiment with others to get a feel of what they do. The following list describes the basic blend factors, which apply to both $\mathbf{F}_{src}$ and $\mathbf{F}_{dst}$. See the **D3D11_BLEND** enumerated type in the SDK documentation for some additional advanced blend factors. Letting $C_{src} = (r_s, g_s, b_s)$, $A_{src} = a_s$ (the RGBA values output from the pixel shader), $\mathbf{C}_{dst} = (r_d, g_d, b_d)$, $A_{dst} = a_d$ (the RGBA values already stored in the render target), $\mathbf{F}$ being either $\mathbf{F}_{src}$ or $\mathbf{F}_{dst}$ and F being either F_{src} or F_{dst}, we have:

D3D11_BLEND_ZERO: F = (0,0,0) and F = 0
D3D11_BLEND_ONE: F = (1,1,1) and F = 1
D3D11_BLEND_SRC_COLOR: F = (r_s, g_s, b_s)
D3D11_BLEND_INV_SRC_COLOR: F = $(1 - r_s, 1 - g_s, 1 - b_s)$
D3D11_BLEND_SRC_ALPHA: F = (a_s, a_s, a_s) and F = a_s
D3D11_BLEND_INV_SRC_ALPHA: F = $(1 - a_s, 1 - a_s, 1 - a_s)$ and F = $1 - a_s$
D3D11_BLEND_DEST_ALPHA: F = (a_d, a_d, a_d) and F = a_d
D3D11_BLEND_INV_DEST_ALPHA: F = $(1 - a_d, 1 - a_d, 1 - a_d)$ and F = $1 - a_d$
D3D11_BLEND_DEST_COLOR: F = (r_d, g_d, b_d)
D3D11_BLEND_INV_DEST_COLOR: F = $(1 - r_d, 1 - g_d, 1 - b_d)$
D3D11_BLEND_SRC_ALPHA_SAT: $\mathbf{F} = (a'_s, a'_s, a'_s)$ and $F = a'_s$
where $a'_s = \text{clamp}(a_s, 0, 1)$
D3D11_BLEND_BLEND_FACTOR: **F** = (r, g, b) and $F = a$, where the color (r, g, b, a) is supplied to the second parameter of the **ID3D11DeviceContext::OMSetBlendState** method (§9.4). This allows you to specify the blend factor color to use directly; however, it is constant until you change the blend state.
D3D11_BLEND_INV_BLEND_FACTOR: **F** = $(1 - r, 1 - g, 1 - b)$ and $F = 1 - a$, where the color (r, g, b, a) is supplied by the second parameter of the **ID3D11DeviceContext::OMSetBlendState** method (§9.4). This allows you to specify the blend factor color to use directly; however, it is constant until you change the blend state.



9.4 BLEND STATE

We have talked about the blending operators and blend factors, but where do we set these values with Direct3D? These blend settings are controlled by the **ID3D11BlendState** interface. Such an interface is found by filling out a **D3D11_BLEND_DESC** structure and then calling **ID3D11Device::CreateBlendState**:

```
HRESULT ID3D11Device::CreateBlendState(
      const D3D11_BLEND_DESC *pBlendStateDesc,
      ID3D11BlendState **ppBlendState);
```

1. **pBlendStateDesc**: Pointer to the filled out **D3D11_BLEND_DESC** structure describing the blend state to create.
2. **ppBlendState**: Returns a pointer to the created blend state interface.

The **D3D11_BLEND_DESC**: Structure is defined like so:

```
typedef struct D3D11_BLEND_DESC {
      BOOL AlphaToCoverageEnable; // Default: False
      BOOL IndependentBlendEnable; // Default: False
      D3D11_RENDER_TARGET_BLEND_DESC RenderTarget[8];

} D3D11_BLEND_DESC;
```

24. What is Direct3d? Explain the resemblance between Direct3D and DirectX?

- Direct3D is a proprietary API by Microsoft that provides functions to render two-dimensional (2D) and three-dimensional (3D) graphics, and uses hardware acceleration if available on the graphics card.
- It was designed by Microsoft Corporation for use on the Windows platform.
- Part of DirectX, Direct3D is used to render three-dimensional graphics in applications where performance is important, such as games.
- Direct3D uses hardware acceleration if it is available on the graphics card, allowing for hardware acceleration of the entire 3D rendering pipeline or even only partial acceleration.
- Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms. Originally, the names of these APIs all began with Direct, such as Direct3D
- The name DirectX was coined as a shorthand term for all of these APIs (the X standing in for the particular API names) and soon became the name of the collection.
- Direct3D (the 3D graphics API within DirectX) is widely used in the development of video games for Microsoft Windows and the Xbox line of consoles. Direct3D is also used by other software applications for visualization and graphics tasks such as CAD/CAM engineering.
- As Direct3D is the most widely publicized component of DirectX, it is common to see the names "DirectX" and "Direct3D" used interchangeably.

25. Explain component object model(com) and any two Interfaces provided by Direct3D?

- COM
    - Component Object Model (COM) is the technology that allows DirectX to be programming language independent and have backwards compatibility.
    - Most of the details of COM are hidden to us when programming DirectX with C++.
        - The only thing that we must know is that we obtain pointers to COM interfaces through special functions or by the methods of another COM interface–we do not create a COM interface with the C++ new keyword.
    - In addition, when we are done with an interface, we call its Release method (all COM interfaces inherit functionality from the IUnknown COM interface, which provides the Release method) rather than delete it
    - COM objects perform their own memory management.

- ID3DXFile interface
    - Applications use the methods of the ID3DXFile interface to create instances of the ID3DXFileEnumObject and ID3DXFileSaveObject interfaces, and to register templates.
    - Members
      The ID3DXFile interface inherits from the IUnknown interface.
    - Methods
      The ID3DXFile interface has these methods.
    - CreateEnumObject
      Creates an enumerator object that will read a .x file.
    - CreateSaveObject
      Creates a save object that will be used to save data to a .x file.
    - RegisterEnumTemplates
      Registers custom templates, given an ID3DXFileEnumObject enumeration object.
    - RegisterTemplates
      Registers custom templates.
    - Remarks
      An ID3DXFile object also contains a local template store. This local storage may be added to only with the ID3DXFile::RegisterEnumTemplates and ID3DXFile::RegisterTemplates methods.
- ID3DXFileEnumObject interface
    - Applications use the methods of the ID3DXFileEnumObject interface to cycle through the child file data objects in the file and to retrieve a child object by its globally unique identifier (GUID) or by its name.
    - Members

The ID3DXFileEnumObject interface inherits from the IUnknown interface.

- Methods
  The ID3DXFileEnumObject interface has these methods.
- GetChild
  Retrieves a child object in this file data object.
- GetChildren
  Retrieves the number of child objects in this file data object.
- GetDataObjectById
  Retrieves the data object that has the specified GUID.
- GetDataObjectByName
  Retrieves the data object that has the specified name.
- GetFile
  Retrieves the ID3DXFile object.
- Remarks
  The GUID for the ID3DXFileEnumObject interface is IID_ID3DXFileEnumObject.
- The LPD3DXFILEENUMOBJECT type is defined as a pointer to this interface.

26. Explain the Input Assembler Stage of rendering Pipeline and explain the vertices and its primitives in detail.

- THE INPUT ASSEMBLER STAGE
  - The input assembler (IA) stage reads geometric data (vertices and indices) from memory and uses it to assemble geometric primitives (e.g., triangles, lines).
- Vertices
  - A vertex in Direct3D can consist of additional data besides spatial location, which allows us to perform more sophisticated rendering effects.
  - We can add normal vectors to our vertices to implement
  - lighting, and texture coordinates to our vertices to implement texturing.
  - Direct3D gives us the flexibility to define our own vertex formats (i.e., it allows us to define the components of a vertex).
- Primitives
  - Vertices are bound to the rendering pipeline in a special Direct3D data structure called a vertex buffer.
  - A vertex buffer just stores a list of vertices in contiguous memory. However, it does not say how these vertices should be put together to form geometric primitives.

  - For example, should every two vertices in the vertex buffer be interpreted as a line or should every three vertices in the vertex buffer be interpreted as a triangle?
  - We tell Direct3D how to form geometric primitives from the vertex data by specifying the primitive topology:
    void ID3D11DeviceContext::IASetPrimitiveTopology(
    D3D11_PRIMITIVE_TOPOLOGY Topology);

27. Explain Rasterization Stage in detail.

- The rasterization stage converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) for the purpose of displaying real-time 3D graphics.
- During rasterization, each primitive is converted into pixels, while interpolating per-vertex values across each primitive.
- Rasterization includes clipping vertices to the view frustum, performing a divide by z to provide perspective, mapping primitives to a 2D viewport, and determining how to invoke the pixel shader.
- While using a pixel shader is optional, the rasterizer stage always performs clipping, a perspective divide to transform the points into homogeneous space, and maps the vertices to the viewport.
- Vertices (x,y,z,w), coming into the rasterizer stage are assumed to be in homogeneous clip-space. In this coordinate space the X axis points right, Y points up and Z points away from camera.
- Viewport Transform
  - After clipping, the hardware can do the perspective divide to transform from homogeneous clip space to normalized device coordinates (NDC).
  - Once vertices are in NDC space, the 2D x- and y- coordinates forming the 2D image are transformed to a rectangle on the back buffer called the viewport.
  - After this transform, the x- and y-coordinates are in units of pixels.
- Backface Culling
  - A triangle has two sides. To distinguish between the two sides we use the following convention.
  - If the triangle vertices are ordered v0, v1, v2 then we compute the triangle normal n like so:
    The side the normal vector emanates from is the front side and the other side is the back side.
  - We say that a triangle is front-facing if the viewer sees the front side of a triangle, and we say a triangle is back-facing if the viewer sees the back side of a triangle.
  - Now, most objects in 3D worlds are enclosed solid objects.
  - Because the front-facing triangles occlude the back-facing triangles, it makes no sense to draw them.

  - Backface culling refers to the process of discarding back-facing triangles from the pipeline.
  - This can potentially reduce the amount of triangles that need to be processed by half.
- Vertex Attribute Interpolation
  - We define a triangle by specifying its vertices. In addition to position, we can attach attributes to vertices such as colors, normal vectors, and texture coordinates.
  - After the viewport transform, these attributes need to be interpolated for each pixel covering the triangle.
  - In addition to vertex attributes, vertex depth values need to get interpolated so that each pixel has a depth value for the depth buffering algorithm.
  - Essentially, interpolation allows us to use the vertex values to compute values for the interior pixels.

30. Explain the following terms with respect to geometry:

a. Angles
b. Intercept Theorems
c. Golden Section
d. Equilateral triangle
e. Circle

10.1.1 Angles

By definition, 360° or 2π [radians] measure one revolution. The reader should be familiar with both units of measurement, and how to convert from one to the other (see page 26). Figure 10.1 shows examples of *adjacent/supplementary* angles (sum to 180°) *opposite* angles (equal), and *complementary* angles (sum to 90°).

10.1.2 Intercept Theorems

Figures 10.2 and 10.3 show scenarios involving intersecting lines and parallel lines that give rise to the following observations:

- First intercept theorem:

$$\frac{a+b}{a} = \frac{c+d}{c}, \quad \frac{b}{a} = \frac{d}{c} \tag{10.1}$$

- Second intercept theorem:

$$\frac{a}{b} = \frac{c}{d} \tag{10.2}$$

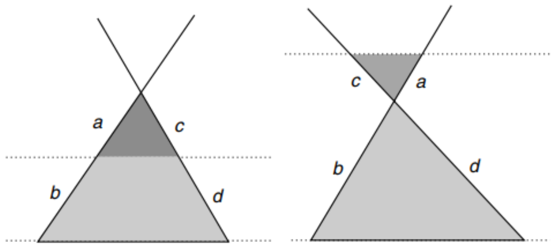


Fig. 10.2. 1st intercept theorem.

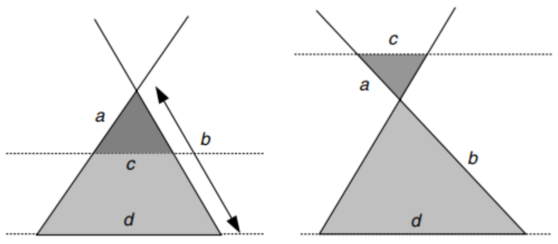


Fig. 10.3. 2nd intercept theorem.

10.1.3 Golden Section

The *golden section* is widely used in art and architecture to represent an 'ideal' ratio for the height and width of an object. Its origins stem from the interaction between a circle and triangle and give rise to the following relationship:

$$b = \frac{a}{2}\left(\sqrt{5} - 1\right) \approx 0.618a \tag{10.3}$$

The rectangle in Figure 10.4 has the proportions

$$height = 0.618 \times width.$$

10.1.7 Equilateral Triangle

An *equilateral* triangle has three equal sides of length l and equal angles of $60°$. The triangle's altitude and area are

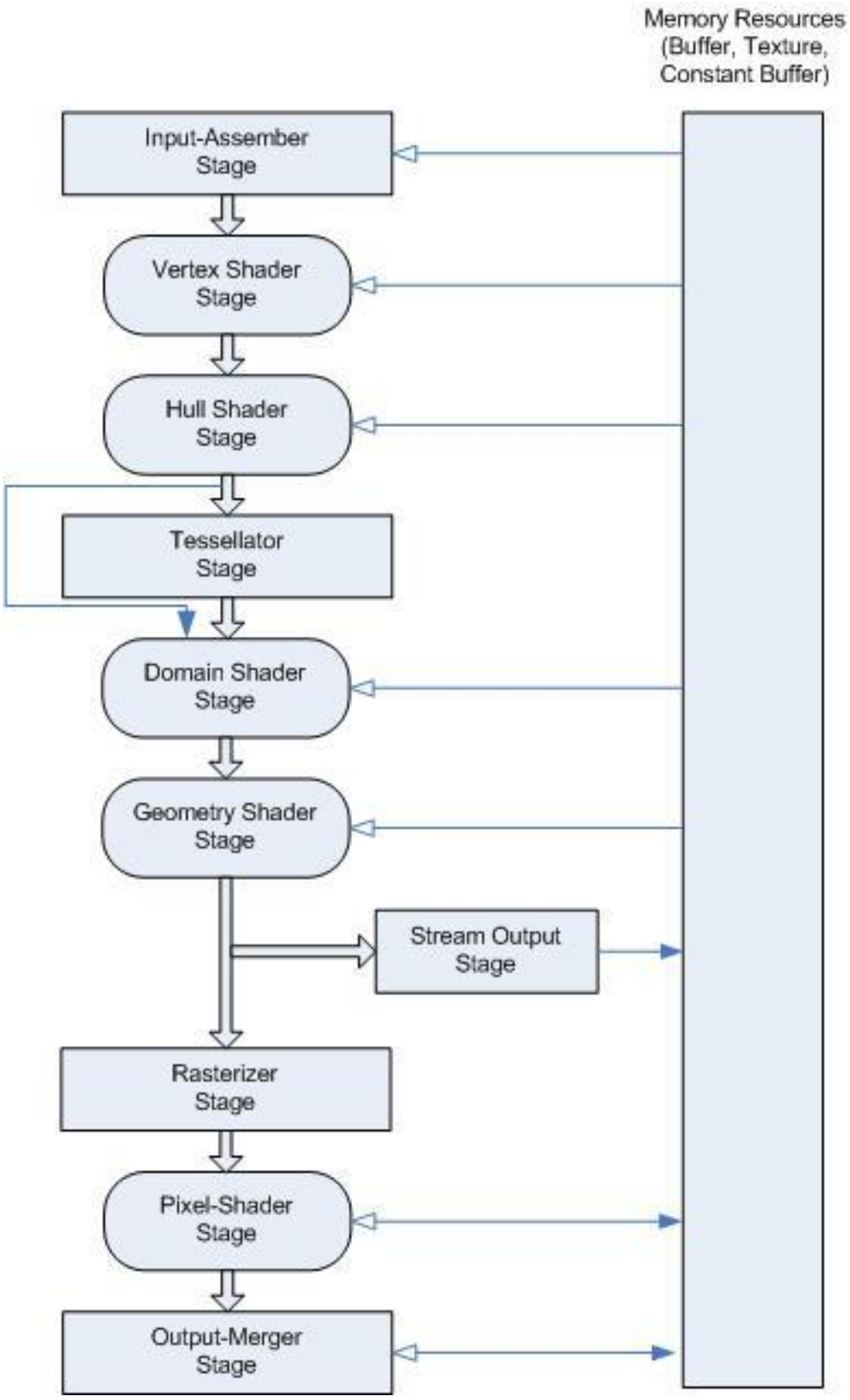
$$h = \frac{\sqrt{3}}{2}l \qquad A = \frac{\sqrt{3}}{4}l^2 \tag{10.9}$$

10.1.16 Circle

The circumference and area of a *circle* are given by

$$C = \pi\, d = 2\pi r \qquad A = \pi\, r^2 = \pi\frac{d^2}{4}$$

31. **Explain in brief the rendering pipeline stages with suitable diagram.**



Input-Assembler Stage	The Direct3D 10 and higher API separates functional areas of the pipeline into stages; the first stage in the pipeline is the input-assembler (IA) stage.
Vertex Shader Stage	The vertex-shader (VS) stage processes vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting. Vertex shaders always operate on a single input vertex and produce a single output vertex. The vertex shader stage must always be active for the pipeline to execute. If no vertex modification or transformation is required, a pass-through vertex shader must be created and set to the pipeline.
Tessellation Stages	The Direct3D 11 runtime supports three new stages that implement tessellation, which converts low-detail subdivision surfaces into higher-detail primitives on the GPU. Tessellation tiles (or breaks up) high-order surfaces into suitable structures for rendering.
Geometry Shader Stage	The geometry-shader (GS) stage runs application-specified shader code with vertices as input and the ability to generate vertices on output.
Stream-Outut Stage	The purpose of the stream-output stage is to continuously output (or stream) vertex data from the geometry-shader stage (or the vertex-shader stage if the geometry-shader stage is inactive) to one or more buffers in memory
Rasterizer Stage	The rasterization stage converts vector information (composed of shapes or primitives) into a raster image (composed of pixels) for the purpose of displaying real-time 3D graphics.
Pixel Shader Stage	The pixel-shader stage (PS) enables rich shading techniques such as per-pixel lighting and post-processing. A pixel shader is a program that combines constant variables, texture data, interpolated per-vertex values, and other data to produce per-pixel outputs. The rasterizer stage invokes a pixel shader once for each pixel covered by a primitive, however, it is possible to specify a **NULL** shader to avoid running a shader.
Output-Merger Stage	The output-merger (OM) stage generates the final rendered pixel color using a combination of pipeline state, the pixel data generated by the pixel shaders, the contents of the render targets, and the contents of the depth/stencil buffers. The OM stage is the final step for determining which pixels are visible (with depth-stencil testing) and blending the final pixel colors.