1

CARTESIAN COORDINATE SYSTEM

Unit Structure :

1.0 OBJECTIVES:

This chapter would make you understand the following concept:

- Use of Cartesian Plane in Graphic
- Representation of various function on graph
- Calculating area of a shapes using graph.
- Euler rule

1.1 INTRODUCTION

Cartesian Co-ordinate system is used to locate the position of a point in a plane using two perpendicular lines. Points are represented in the form of coordinates (x, y) in two-dimension with respect to x- and y- axes.

A Cartesian coordinate system in two dimensions is commonly defined by two axes, at right angles to each other, forming a plane (an *xy*-plane). The horizontal axis is normally labelled *x*, and the vertical axis is normally labelled *y*. In a three–dimensional coordinate system, another axis, normally labelled *z*, is added, providing a third dimension of space measurement. A plane consists of axes and quadrants. Thus, we call the plane the Cartesian Plane, or the Coordinate Plane, or the Cartesian x-y plane. The axes are called the coordinate axes.The fig1.1 shows the cartesian coordinate system with four quadrants.

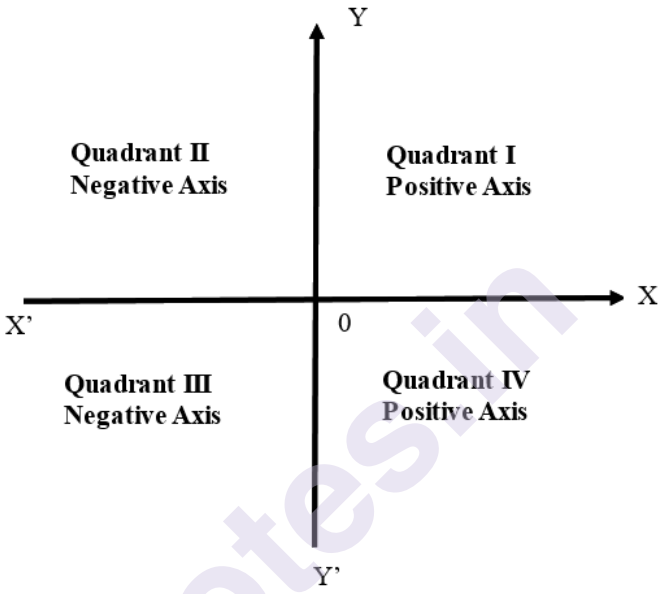


Fig:1.1 Cartesian Coordinate System with four Quadrant

1.2 The Cartesian XY-plane

- The Cartesian xy-plane provides a mechanism for translating variables (Paired variables) into a graphical format.
- The variables are normally x and y that are used to describe a function such as:-

  y = 3x+2.

- Every value of x has a corresponding value of y.

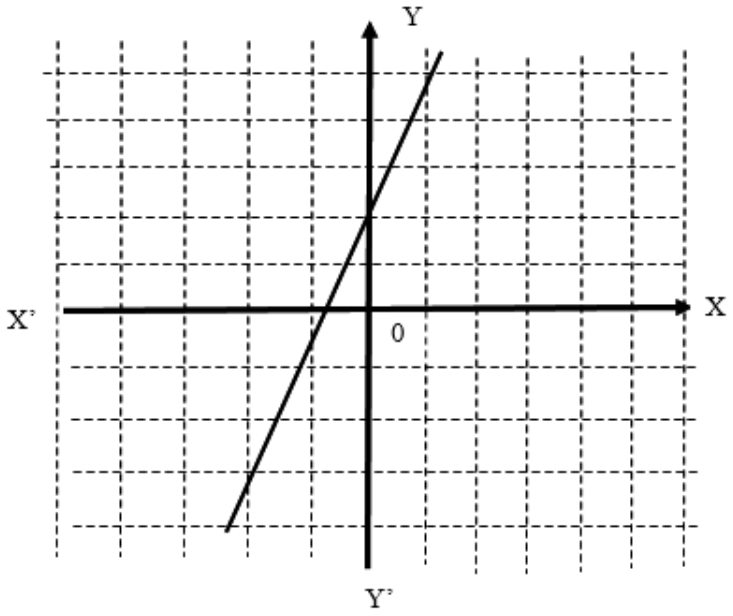


Fig:1.2The equation y = 3x + 2 using the xy Cartesian plane.

- A Cartesian XY plane consists of axes and quadrants in the cartesian coordinate system.
- Descartes suggested that the letters x and y should be used to represent variables, and letters at the other end of the alphabet should substitute numbers. That is why equations such as y = ax2 + bx + c is written the way as it is.
- By convention, in cartesian coordinate system, the axis for the independent variable x is horizontal, and the dependent variable y is vertical. The axes intersect at 90◦ at a point called the origin.
- Measurements to the right and left of the origin are positive and negative respectively, and measurements above and below the origin share a similar sign convention. Together, the axes are said to create a left-handed set of axes, because it is possible, using one's left hand, to align the thumb with the x -axis and the first finger with the y-axis.
- Any point P on the Cartesian plane is identified by an ordered pair of numbers (x, y) where x and y are called the Cartesian coordinates of P.
- Mathematical functions and geometric shapes can then be represented as lists of coordinates inside a program.

1.2.1 Function Graphs

- A Different type of functions, such as

  y = mx + c (linear function),

  $y = ax^2 + bx + c$ (quadratic function),

$y = ax^3 + bx^2 + cx + d$ (cubic),

$y = a \sin(x)$ (trigonometric), etc.

will create familiar shapes that permit the function to be easily identified.

- Linear functions are straight lines, quadratics are parabolas, cubic will have an 's' shape, and trigonometric functions will have a wave-like trace.
- Fig: 1.3 Shows examples of each type of function.

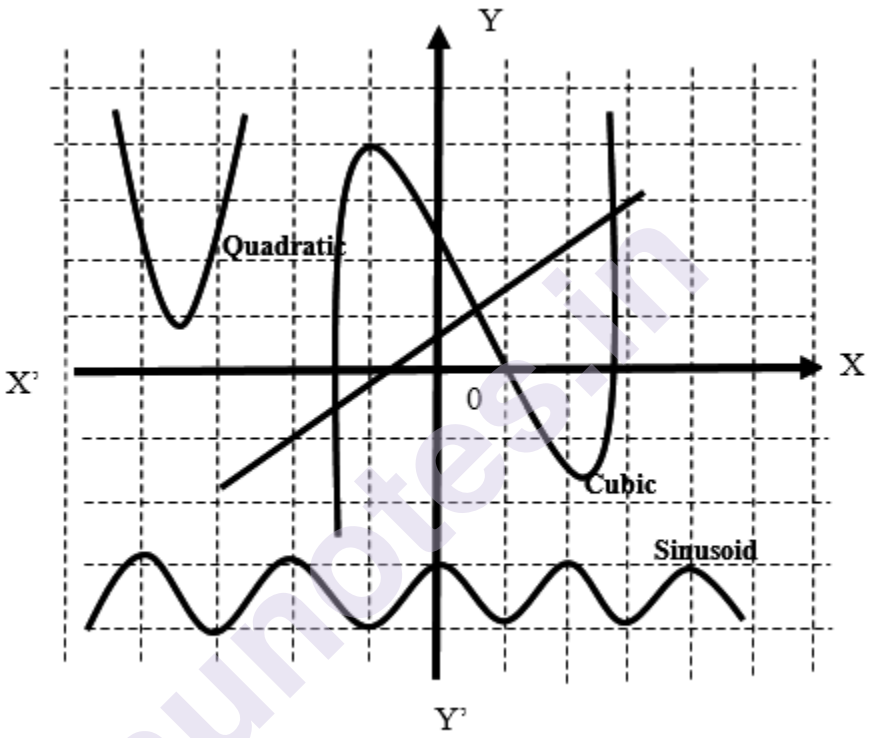


Fig: 1.3: Graph of four function type.

- Such graphs are used in computer animation to control the movement of objects, lights and the virtual camera.
- But instead of describing the relationship between x and y, the graphs show the relationship between an activity such as movement, rotation, size, brightness, colour, etc., with time. Figure 1.4 shows an example where the horizontal axis marks the progress of time in animation frames, and the vertical axis records the corresponding brightness of a virtual light source.

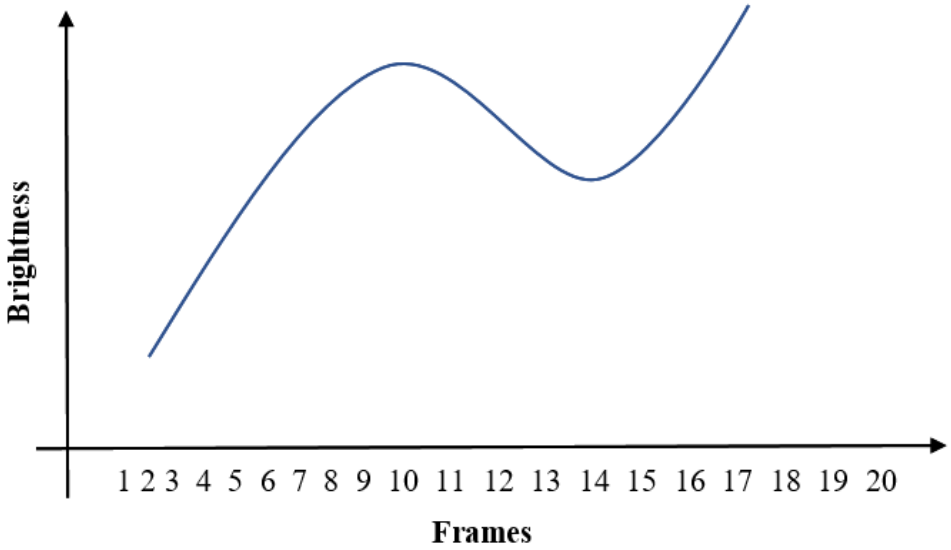


Fig:1.4: Graph showing relationship between Brightness and frames

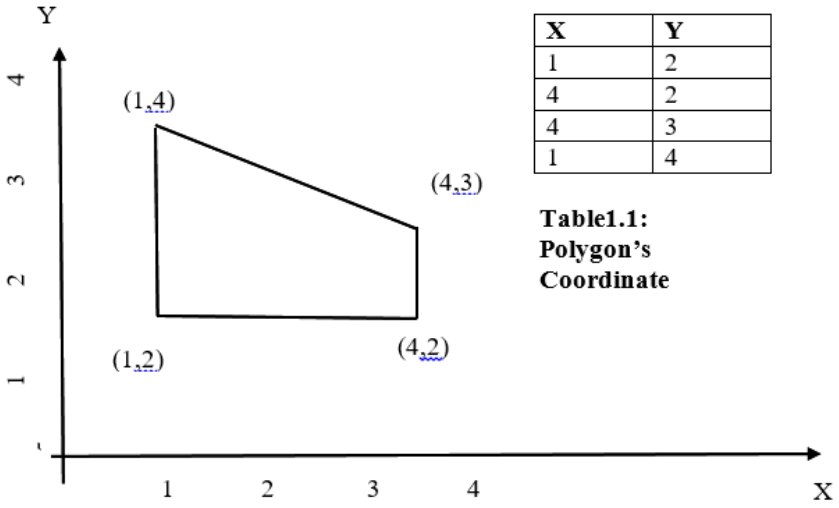
Such a graph helps animator to make changes to the function with the aid of interactive software tools and achieve appropriate animation.

1.2.2 Geometric Shapes

- Computer graphics requires that 2D shapes and 3D objects have a numerical description of some sort.
- Shapes can include polygons, circles, arbitrary curves, mathematical functions, fractals, etc., and objects can be faceted, smooth, bumpy, furry, gaseous, etc.
- The Cartesian plane also provides a way to represent 2D shapes numerically, which permits them to be manipulated mathematically.

1.2.3 Polygonal Shapes

- A polygon is constructed from a sequence of vertices (points) as shown in Figure 1.5.
- A straight line is assumed to link each pair of neighbouring vertices; intermediate points on the line are not explicitly stored.
- There is no convention for starting a chain of vertices, but software will often state whether polygons have a clockwise or anti-clockwise vertex sequence.



X	Y
1	2
4	2
4	3
1	4

Table1.1: Polygon's Coordinate

Fig:1.5 A simple polygon created with four vertices shown in the

- If the vertices in Figure 1.5 had been created in an anti-clockwise sequence, they could be represented in a tabular form as shown in the above table 1.1, where the starting vertex is (1, 1), but this is arbitrary.
- We can now perform various arithmetic and mathematical operations on this list of vertex coordinates.
- For example, if we double the values of x and y and redraw the vertices, we discover that the form of the shape is preserved, but its size is doubled with respect to the origin.
- Similarly, if we divide the values of x and y by 2, the shape is still preserved, but its size is halved with respect to the origin.
- On the other hand, if we add 1 to every x -coordinate and 2 to every y-coordinate and redraw the vertices, the shape's size remains the same but it is moved 1 unit ahead horizontally and 2 units ahead vertically.

1.2.4 Area of a Shape

- The area of a polygonal shape is readily calculated from its list of coordinates. For example, using the list of coordinates shown in Table 1.2 : the area is computed by

X	Y
X0	Y0
X1	Y1
X2	Y2
X3	Y3

Table 1.2 Polygon's Coordinates

Area=1/2 [(x0y1 − x1y0) + (x1y2 − x2y1) + (x2y3 − x3y2) + (x3y0 − x0y3)]

- You will observe that the calculation sums the results of multiplying an x by the next y, minus the next x by the previous y. When the last vertex is selected, it is paired with the first vertex to complete the process. The result is then halved to reveal the area.

- As a simple test, let's apply this formula to the shape described in Fig. 1.5:

Area=1/2 [(1 x2 −4 x 2) + (4 x 3 – 4 x 2) + (4 x 4 – 1 x 3) + (1 x 2 – 1 x 4)]

=1/2[-6 + 4 + 13 - 2]

=4.5

- The beauty of this technique is that it works with any number of vertices and any arbitrary shape.

- Another feature of this technique is that if the original set of coordinates is clockwise, the area is negative. Which means that the calculation computes vertex sequence as well as area. To illustrate this feature, consider the below table for the above fig: 1.5 with list of polygon's coordinates in clockwise sequence:

X	Y
1	2
1	4
4	3
4	2

Area=1/2 [(x0y1 − x1y0) + (x1y2 − x2y1) + (x2y3 − x3y2) + (x3y0 − x0y3)]

=1/2[(1 x 4 - 1 x 2) + (1 x 3 - 4 x 4) + (4 x 2 – 4 x 3) + (4 x 2 -1 x 2)]

=1/2[2 -13 -4 + 6]

= -4.5

The minus sign indicates that the vertices are in a clockwise sequence.

1.2.5 Theorem of Pythagoras in 2D

- Pythagoras proved that the **squared length of a** plus **the squared length of b** equals the **squared length of c**, if a, b and c form a triangle where angle ab is 90°.

- This results in the equation:

$$a^2 + b^2 = c^2$$

Solving it for c we will get

$$c = \sqrt{(a^2 + b^2)}$$

- We can calculate the distance between two points by applying the theorem of Pythagoras.

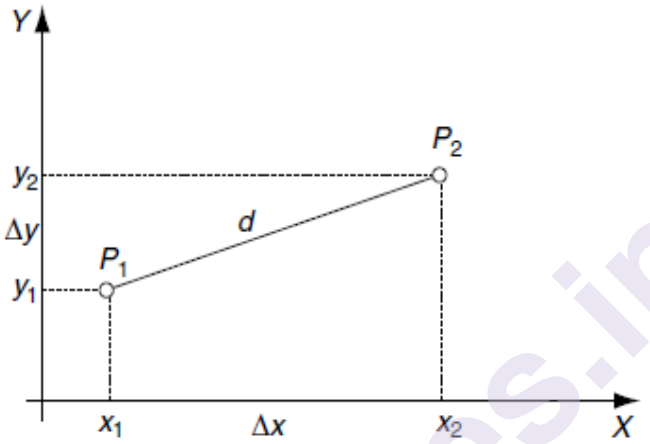


Fig:1.6: Calculating the distance between two

Figure 1.6 shows two arbitrary points P1(x1, y1) and P2(x2, y2). The distance Δx = x2−x1 and Δy = y2−y1. Therefore, the distance d between P1 and P2 is given by

$d = \Delta x^2 + \Delta y^2$.

1.3 3D COORDINATES

- In the 2D Cartesian plane a point is located by its x - and y-coordinates.
- But when we move to 3D there are two ways in which the third z-axis can be positioned.
- Figure 1.6 shows the two ways, which are described as left- and right handed axial systems.

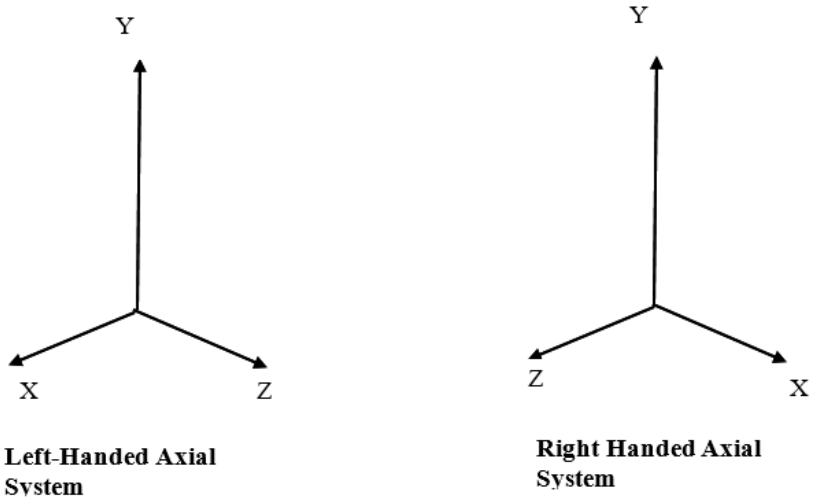


Fig:1.6 Two Axial System in 3D

- The left-handed system allows us to align our left hand with the axes such that the thumb aligns with the x -axis, the first finger aligns with the y-axis and the middle finger aligns with the z -axis.
- The right-handed system allows the same system of alignment, but using our right hand.
- The choice between these axial systems is arbitrary, but one should be aware of the system employed by commercial computer graphics packages.

1.3.1 Theorem of Pythagoras in 3D

- The theorem of Pythagoras in 3D is a natural extension of the 2D rule.
- It is also applicable to higher dimensions.
- Given two arbitrary points P1(x1, y1, z1) and P2(x2, y2, z2),

  the distance $\Delta x = x2 - x1$, $\Delta y = y2 - y1$ and $\Delta z = z2 - z1$.

  Therefore, the distance d between P1 and P2 is given by

  $d = \sqrt{(\Delta x^2 + \Delta y^2 + \Delta z^2)}$

1.3.2 3D Polygons

- The simplest 3D polygon is a triangle, which is always planar, i.e., the three vertices lie on a unique plane.
- Planarity is very important in computer graphics because rendering algorithms assume that polygons are planar.
- For instance, it is quite easy to define a quadrilateral in 3D where the vertices are not located on one plane. When such a polygon is rendered (presented) and animated, improper highlights can result,

simply because the geometric techniques (which assume the polygon is planar) give rise to errors.

1.3.3 Euler's Rule

- In 1619, Descartes discovered relationship between vertices, edges and the faces of a 3D polygonal object.
- According to him, **faces + vertices = edges + 2**.
- For example, consider a cube;
- it has 12 edges, 6 faces and 8 vertices, which satisfies this equation.
- This rule can be applied to a geometric database to discover whether it contains any false features.
- Unfortunately for Descartes, for some unknown reason, the rule is named after Euler

1.4 SUMMARY

The Cartesian plane and its associated coordinates are the basis for all mathematics used for computer graphics. Shapes can be manipulated using simple functions, and the plane can be extended into a 3D Cartesian space that becomes the domain for creating objects, curves, surfaces, and a virtual environment where they can be animated and visualized.

1.5 QUESTION

1) Explain in detail the Cartesian xy-plane.
2) Write a short note on Theorem of Pythagoras in 2D.
3) Write a short note on Theorem of Pythagoras in 3D.
4) Explain Euler's Rule with suitable example.
5) Describe cartesian xy plane and explain the concept of function graph.

1.6 REFERENCES

Mathematics for Computer Graphics, John Vince, Springer-Verlag London,2nd Edition.

❄❄❄❄❄❄❄

2

VECTOR

Unit Structure :

2.0 OBJECTIVES:

This chapter would make you understand the following concept:

- Basic operations on vector.
- Use of dot product and cross product in computer graphics

- Power of unit vector in calculation
- Position vector
- Cartesian Vectors

2.1 INTRODUCTION:

- Vectors are a relatively new arrival to the world of mathematics, dating only from the 19th century.
- **Vectors, in Maths,** are objects which have both, magnitude and direction. Magnitude defines the size of the vector. It is represented by a line with an arrow, where the length of the line is the magnitude of the vector and the arrow shows the direction.
- Vectors provide us with some elegant and powerful techniques for computing angles between lines and the orientation of surfaces.
- They also provide a clear framework for computing the behaviour of dynamic objects in computer animation and illumination models in rendering.
- We always use single number to represent quantities such as, height, age, shoe size, waist and chest measurements. Such quantities are called **scalars.**
- In computer graphics scalar quantities include colour, height, width, depth, brightness, number of frames, etc.
- On the other hand, there are some things such as wind, force, weight, velocity and sound etc, that require more than one number to represent them.
- For example, any sailor knows that wind has a magnitude and a direction. The force we use to lift an object also has a value and a direction. Similarly, the velocity of a moving object is measured in terms of its speed (e.g., miles per hour) and a direction such as north-west. Sound, too, has intensity and a direction. These quantities are called vectors.
- In computer graphics, vectors are generally made of two or three numbers.
- Mathematicians such as Caspar Wessel (1745–1818), Jean Argand (1768– 1822) and John Warren (1796–1852) were simultaneously exploring complex numbers and their graphical representation. In 1837, Sir William Rowan Hamilton (1788–1856) made his breakthrough with quaternions. In 1853, Hamilton published his book Lectures on Quaternions in which he described terms such as vector, transvector and provector. Hamilton's work was not widely accepted until 1881, when the American mathematician Josiah Gibbs (1839–1903) published his treatise Vector Analysis, describing modern vector analysis.

2.2 2D VECTORS

- In computer graphics we use 2D and 3D vectors.
- It is a vector in 2D space.

2.2.1 Vector Notation

- A scalar such as x is a name for a single numeric quantity.
- However, because a vector contains two or more numbers, its symbolic name is printed using a bold font to make it different from a scalar variable.

Examples are **n**, **i** and **Q**.

- When a scalar variable is assigned a value, we use the standard algebraic notation

  $x = 3$

- However, when a vector is assigned its numeric values, the following notation is used:

  $\mathbf{n} = \begin{matrix} 3 \\ 2 \end{matrix}$ which is called a column vector.

- The numbers 3 and 2 are called the components of **n**, and their position within the brackets is significant.
- A row vector transposes the components horizontally, $\mathbf{n} = [3\ 2]^T$,where the superscript T means transposition.

2.2.2 Graphical Representation of Vectors

- As Vectors have to express direction as well as magnitude, an arrow could be used to indicate direction and a number can be used to specify magnitude.
- Cartesian coordinates provide an excellent mechanism for visualizing vectors and allowing them to be included within the classical framework of mathematics.
- Figure 2.1 shows a vector represented by a short line segment. The length of the line represents the vector's magnitude, and its orientation defines its direction. But as we can see from the figure, the line does not have a direction. Even if we attach an arrowhead to the line, which is standard practice for annotating vectors in books and scientific papers, the arrowhead has no mathematical reality.

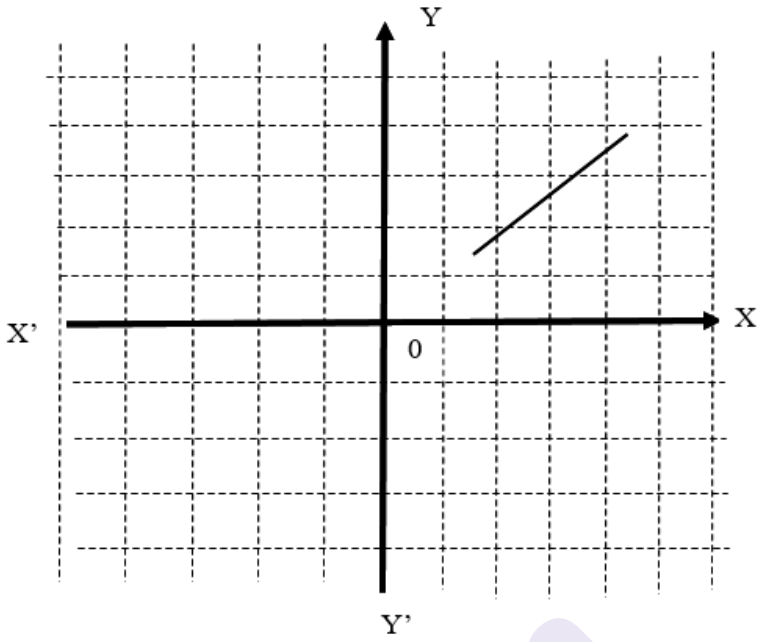


Fig:2.1: Graphical representation of Vector

- The line's direction can be determined by first identifying the vector's tail and then measuring its components along the x - and y-axes.

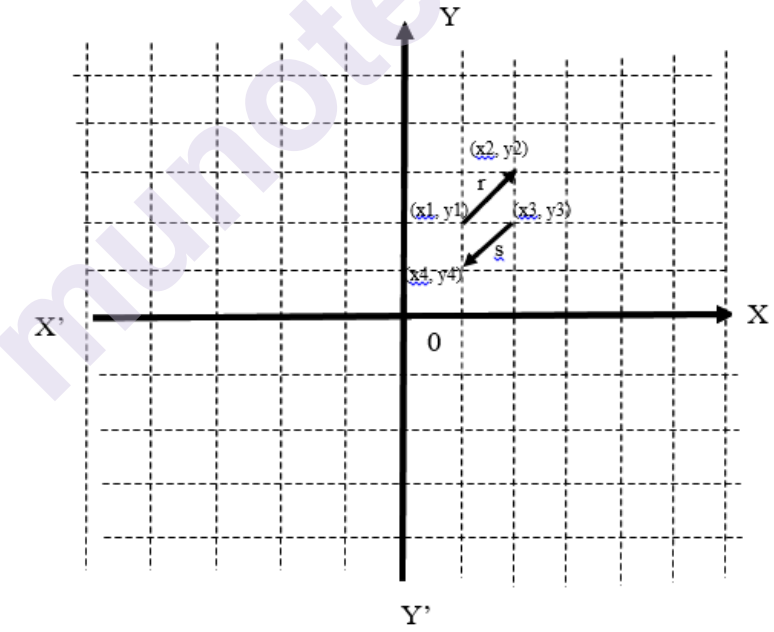


Fig:2.2 Two vectors r and s have the same magnitude and opposite directions

- For example, in Figure 2.2 the vector **r** has its tail defined by (x1, y1) = (1, 2) and its head by (x2, y2) = (2, 3).
- Vector **s**, on the other hand, has its tail defined by (x3, y3) = (2, 2) and its head by (x4, y4) = (1, 1).
-

- The x - and y-components for **r** are computed as follows:

$$x_r = (x2 - x1) \quad x_r = 2 - 1 = 1$$

$$y_r = (y2 - y1) \quad y_r = 3 - 2 = 1$$

- whereas the components for **s** are computed as follows:

$$x_s = (x4 - x3) \quad x_s = 1 - 2 = -1$$

$$y_s = (y4 - y3) \quad y_s = 1 - 2 = -1$$

$$x_s = -1 \text{ and } y_s = -1$$

The negative value of x_s and y_s shows the direction of the vector **s**.

In general, if the coordinates of a vector's head and tail is given by $(x_h, y_{h)}$ and $(x_t, y_{t)}$ respectively, then its components Δx and Δy are given by

$$\Delta x = (x_h - x_t)$$

$$\Delta y = (y_h - y_t)$$

One can readily see from this notation that a vector does not have a unique position in space. It does not matter where we place a vector: so long as we preserve its length and orientation, its components will not alter.

2.2.3 Magnitude of a Vector

- The magnitude of a vector **r** is expressed by $\|r\|$ and is computed by applying the theorem of Pythagoras to its components:

$$\|r\| = \Delta x^2 + \Delta y^2$$

To illustrate this idea, consider a vector defined by $(x_h, y_h) = (3, 4)$ and $(x_t, y_t) = (1, 1)$.

The x - and y-components are 2 and 3 respectively.

Therefore, its magnitude is equal to

$\sqrt{22} + 32 = 3.606$

2.3 3D VECTORS

- It is extremely simple to extend the notation of 2D vector to include an extra dimension. Figure 2.3 shows a 3D vector r with its head, tail, components and magnitude annotated.

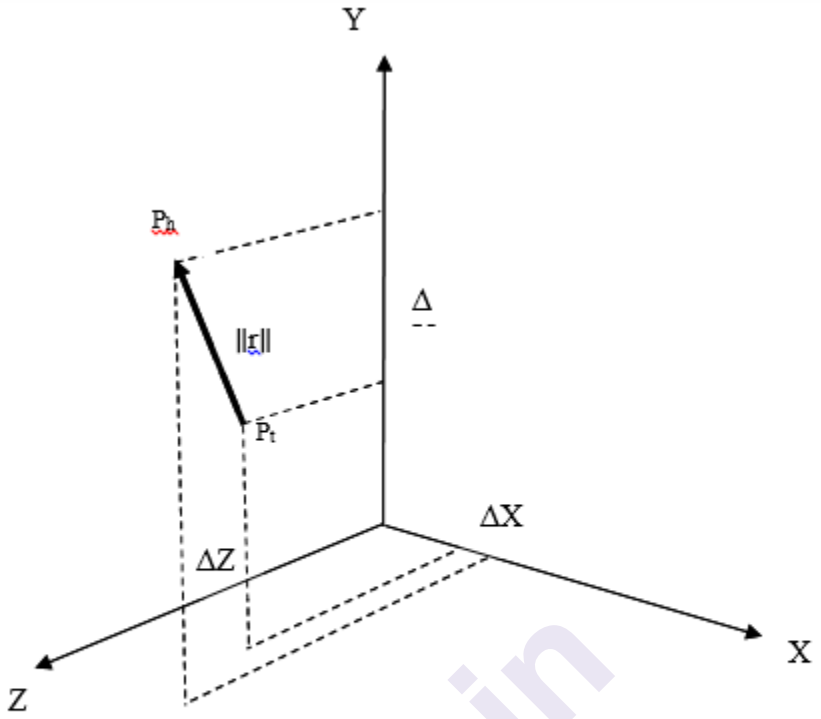


Fig:2.3 The 3D vector with its components Δx, Δy, Δz, which are the differences between the head and tail coordinates.

- As it is a 3D vector, it will be having 3 components, i.e., Δx, Δy and Δz.
- The components and magnitude of a 3D vector are given by

  $\Delta x = (x_h - x_t)$

  $\Delta y = (y_h - y_t)$

  $\Delta z = (z_h - z_t)$

  $||r|| = \sqrt{(\Delta x^2 + \Delta y^2 + \Delta z^2)}$

2.3.1 Vector Manipulation

- As vectors are different from scalars, a set of rules has been developed to control how the two mathematical entities interact with one another.
- For example, we need to consider vector addition, subtraction and multiplication, and how a vector can be modified by a scalar.
- Vector manipulation is the power to manipulate the properties of objects described via vectors by modifying these vectors directly.

2.3.1.1 Multiplying a Vector by a Scalar

- When a vector is multiplied by a positive scalar quantity, then the magnitude of the vector changes in accordance with the magnitude of the scalar but the direction of the vector remains unchanged.
- But if the vector is multiplied by a negative scalar quantity, then the direction of the vector will be just opposite to the original direction.
- Given a vector **n**, 2**n** means that the vector's components are doubled.
- For example, if **n** = $\begin{bmatrix}3\\6\\5\end{bmatrix}$ then $2\mathbf{n} = \begin{bmatrix}6\\12\\10\end{bmatrix}$ which seems logical.
- Similarly, if we **divide n** by 2, its components are halved.

2.3.1.2 Vectors Addition and Subtraction

Given two vectors **r** and **s**, **r ± s** is defined as :-

r= $\begin{bmatrix}xr\\yr\\zr\end{bmatrix}$ **s**= $\begin{bmatrix}xs\\ys\\zs\end{bmatrix}$ **r ± s**== $\begin{bmatrix}xr \pm xs\\yr \pm ys\\zr \pm zs\end{bmatrix}$

Vectors addition is commutative in nature: i.e., a + b = b + a

E.g.: $\begin{bmatrix}3\\6\\5\end{bmatrix} + \begin{bmatrix}2\\1\\4\end{bmatrix} = \begin{bmatrix}5\\7\\9\end{bmatrix}$ And $\begin{bmatrix}2\\1\\4\end{bmatrix} + \begin{bmatrix}3\\6\\5\end{bmatrix} = \begin{bmatrix}5\\7\\9\end{bmatrix}$

However, like scalar subtraction, vector subtraction is not commutative. a − b ≠ b − a

E.g.: $\begin{bmatrix}3\\6\\5\end{bmatrix} - \begin{bmatrix}2\\1\\4\end{bmatrix} = \begin{bmatrix}1\\5\\1\end{bmatrix}$ And $\begin{bmatrix}2\\1\\4\end{bmatrix} - \begin{bmatrix}3\\6\\5\end{bmatrix} = \begin{bmatrix}-1\\-5\\-1\end{bmatrix}$

So, a − b ≠ b – a.

Position Vectors

- Given any point P (x, y, z), a position vector **p** can be created by assuming that P is the vector's head and the origin is its tail.
- In other words, position vector is a vector whose tail is the origin. That is the coordinates of the tail of the position vector will be (0,0,0).
- Because the tail coordinates are (0, 0, 0), the position vector's components are x, y, z.
- Consequently, the position vector's magnitude ||p|| will be equal to $\sqrt{(x^2 + y^2 + z^2)}$.

- For example, the point P(4, 5, 6) creates a position vector **p** relative to the origin:

$$p = \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \quad \text{and} \quad ||p|| = \sqrt{(4^2 + 5^2 + 6^2)} = 20.88$$

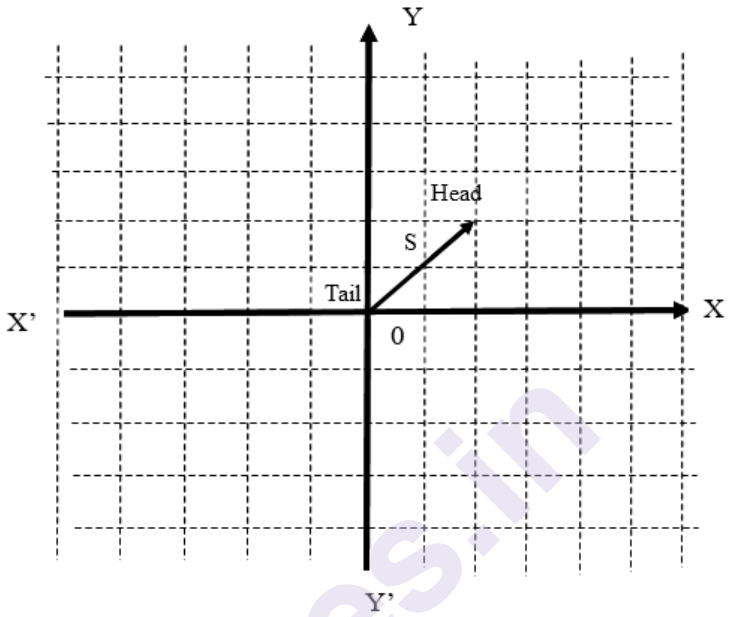


Fig:2.4: Graphical representation of Position Vector S

- The figure 2.4 show a position vector **S** whose tail is its origin and coordinates of head is (2,2).

2.3.3 Unit Vectors

- By definition, a unit vector has a magnitude of 1.
- A simple example is **i** , where

$$\mathbf{i} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \qquad ||\mathbf{i}|| = 1$$

- Unit vectors are extremely useful when we come to vector multiplication.
- It is because multiplication of vectors involves taking their magnitude, and if this is unity, the multiplication is greatly simplified.
- Furthermore, in computer graphics applications, vectors are used to specify the orientation of surfaces, the direction of light sources and the virtual camera. Again, if these vectors have a unit length, the computation time associated with vector operations can be minimized.

- Converting a vector into a unit form is called **normalizing** and is achieved by dividing a vector's components by its magnitude.
- To formalize this process, consider a vector **r** whose components are x, y, z.

The magnitude $||r||=\sqrt{(x^2+y^2+z^2)}$

And the unit form of r are given by

$$r_u = \frac{1}{\|r\|}\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Consider the conversion of r into a unit form :

$$r=\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}$$

$|| r|| = \sqrt{1^2 + 2^2 +3^2)}=\sqrt{14}$

$$r_u=\frac{1}{\sqrt{14}}\begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}=\begin{bmatrix} 0.267 \\ 0.535 \\ 0.802 \end{bmatrix}$$

2.3.4 Cartesian Vectors

- We have studied the scalar multiplication of vectors, vector addition and unit vectors.
- we can combine all three to permit the algebraic manipulation of vectors.
- To begin with, we will define three Cartesian unit vectors **i, j, k** that are aligned with the x -, y- and z -axes respectively.

$$\mathbf{i}=\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \quad \mathbf{j}=\begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix} \quad \mathbf{k}=\begin{bmatrix} 0 \\ 0 \\ 1 \end{bmatrix}$$

- Therefore, any vector aligned with the x-, y- or z -axes can be defined by a scalar multiple of the unit vectors **i, j and k** respectively.
- For example, a vector 10 unit long aligned with the x -axis is simply 10**i**, and a vector 20 units long aligned with the z -axis is 20**k**.
- By employing the rules of vector addition and subtraction, we can compose a vector **r** by adding three Cartesian vectors as follows:

  r = a**i** + b**j**+ c**k**

This is equivalent to writing **r** as

$$\mathbf{r} = \begin{bmatrix} a \\ b \\ c \end{bmatrix}$$

which means that the magnitude of **r** is readily computed as

$||r|| = \sqrt{(a^2 + b^2 + c^2)}$

Any pair of Cartesian vectors such as **r** and **s** can be combined as follows:

r = a**i** + b**j** + c**k**

s = d**i** + e**j** + f**k**

r ± s = (a ± d) **i** + (b ± e) **j** + (c ± f) **k**

For example, given

r = 3**i** + 2**j** + 4**k** and **s** = 2**i** + 5**j** + 6**k**

then

r + **s** = 5**i** + 7**j** + 10**k**

and

$||\mathbf{r} + \mathbf{s}|| = \sqrt{(5^2 + 7^2 + 10^2)} = \sqrt{174} = 13.19$

2.3.5 Vector Multiplication

- Although vector addition and subtraction are useful in resolving various problems, vector multiplication provides some powerful ways of computing angles and surface orientations.
- The multiplication of two scalars is very familiar: for example, 6×7 or 7× 6 = 42.
- However, when we consider the multiplication of vectors, we are basically multiplying two 3D lines together, which is not an easy operation to visualize.
- Mathematicians have discovered that there are two ways to multiply vectors together: one gives rise to a scalar result and the other give rise to a vector result.
- When the multiplication of two vectors give rise to a scalar result then it is known as the scalar product.
- When the multiplication of two vectors give rise to a vector result then it is known as the vector product.

2.3.5.1 Scalar Product

- We could multiply two vectors **r** and **s** by using the product of their magnitudes:

  $||\mathbf{r}|| \cdot ||\mathbf{s}||$.

- Although this is a valid operation, it does not get us anywhere because it ignores the orientation of the vectors, which is one of their important features.

- But this concept is developed into a useful operation by including the angle between the vectors.

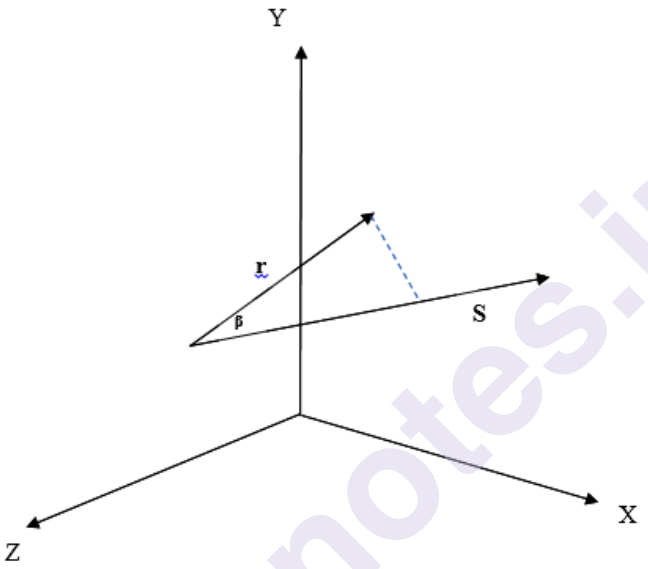


Fig:2.5 The projection of r on s creates the basis for the scaler product.

- Figure 2.5 shows two vectors **r** and **s** that have been drawn, for convenience, such that their tails touch.

- Taking **s** as the reference vector, which is an arbitrary choice, we compute the projection of **r** on **s**, which takes into account their relative orientation.

- The length of **r** on **s** is

  $||\mathbf{r}|| \cos(\beta)$.

  We can now multiply the magnitude of **s** by the projected length of **r**:

  $||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta)$.

  This scalar product is written

  $\mathbf{s} \cdot \mathbf{r} = ||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta)$

- The dot symbol '·' is used to represent scalar multiplication, to distinguish it from the vector product.
- Because of this symbol, the scalar product is often referred to as the **dot product**.
- To compute dot product, we define two Cartesian vectors **r** and **s**, and proceed to multiply them together using the dot product definition:

  $\mathbf{r} = a\mathbf{i} + b\mathbf{j} + c\mathbf{k}$

  $\mathbf{s} = d\mathbf{i} + e\mathbf{j} + f\mathbf{k}$

therefore

$\mathbf{r} \cdot \mathbf{s} = (a\mathbf{i} + b\mathbf{j} + c\mathbf{k}) \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) = a\mathbf{i} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) + b\mathbf{j} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k}) + c\mathbf{k} \cdot (d\mathbf{i} + e\mathbf{j} + f\mathbf{k})$

$\mathbf{r} \cdot \mathbf{s} = ad\,(\mathbf{i} \cdot \mathbf{i}) + ae\,(\mathbf{i} \cdot \mathbf{j}) + af\,(\mathbf{i} \cdot \mathbf{k}) + bd\,(\mathbf{j} \cdot \mathbf{i}) + be\,(\mathbf{j} \cdot \mathbf{j}) + bf\,(\mathbf{j} \cdot \mathbf{k}) + cd\,(\mathbf{k} \cdot \mathbf{i}) + ce\,(\mathbf{k} \cdot \mathbf{j}) + cf\,(\mathbf{k} \cdot \mathbf{k})$

Using the definition of the dot product, terms such as $(\mathbf{i} \cdot \mathbf{i})$, $(\mathbf{j} \cdot \mathbf{j})$ and $(\mathbf{k} \cdot \mathbf{k}) = 1$, because the angle between **i** and **i**, **j** and **j**, or **k** and **k** is 0◦; and cos (0◦) = 1.

But because the other vector combinations are separated by 90◦, and cos (90◦) = 0, all remaining terms will be equal to zero.

Bearing in mind that the magnitude of a unit vector is 1, we can write

$||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta) = ad + be + cf$

This result confirms that the dot product is indeed a scalar quantity.

2.3.5.1.1 Example of the Dot Product

To find the angle between two vectors **r** and **s**,

$\mathbf{r} = \begin{bmatrix} 2 \\ -3 \\ 4 \end{bmatrix}$ and $\mathbf{s} = \begin{bmatrix} 5 \\ 6 \\ 10 \end{bmatrix}$

$||\mathbf{r}|| = \sqrt{(2^2 + (-3)^2 + 4^2)} = 5.385$ and

$||\mathbf{s}|| = \sqrt{(5^2 + 6^2 + 10^2)} = 12.689$

Therefore

$||\mathbf{s}|| \cdot ||\mathbf{r}|| \cos(\beta) = 2 \times 5 + (-3) \times 6 + 4 \times 10 = 32$

$12.689 \times 5.385 \times \cos(\beta) = 32$

$\cos(\beta) = \frac{32}{12.689 \times 5.385} = 0.468$

$\beta = \cos^{-1}(0.468) = 62.1^{\circ}$

The angle between the two vectors is 62.1°

2.3.5.1.2 The Dot Product in Lighting Calculations

- 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.
- This is shown in Figure 2.6. The light source is located at (20, 20, 40) and the illuminated point is (0, 10, 0). In this situation we are interested in calculating cos(β), which when multiplied by the light source intensity gives the incident light intensity on the surface.

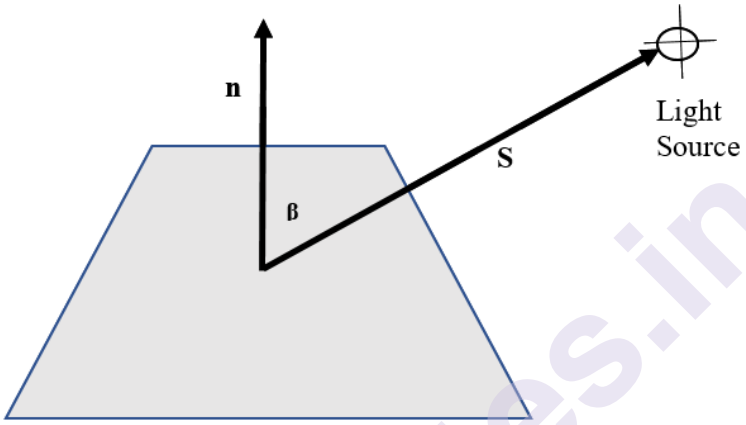


Fig:2.6 : . Lambert's law representation

- To begin with, we are given the normal vector **n** to the surface. In this case **n** is a unit vector, and its magnitude $\|\mathbf{n}\| = 1$.

$$\mathbf{n} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

The direction of the light source from the surface is defined by the vector s:

$$\mathbf{s} = \begin{bmatrix} 20 - 0 \\ 20 - 10 \\ 40 - 0 \end{bmatrix} = \begin{bmatrix} 20 \\ 10 \\ 40 \end{bmatrix}$$

$\|\mathbf{s}\| = \sqrt{(20^2 + 10^2 + 40^2)} = 45.826$

$\|\mathbf{n}\| \cdot \|\mathbf{s}\| \cos(\beta) = 0 \times 20 + 1 \times 10 + 0 \times 40 = 10$

$1 \times 45.826 \times \cos(\beta) = 10$

$\cos(\beta) = \frac{10}{45.826} = \mathbf{0.218}$

Therefore, the light intensity at the point (0, 10, 0) is **0.218** of the original light intensity at

(20, 20, 40).

2.3.5.1.3 The Dot Product in Back-Face Detection

- Back-face detection means determination of whether a face of an object is facing backward and therefore that face is invisible.
- A standard way of identifying back-facing polygons relative to the virtual camera is to compute the angle between the polygon's surface normal and the line of sight between the camera and the polygon.
- If this angle **is less than 90◦ the polygon is visible**.
- If it is **equal to or greater than 90◦ the polygon is invisible**.
- An Example is shown in Figure 2.7. It is clear from the figure that the right-hand polygon is invisible to the camera,

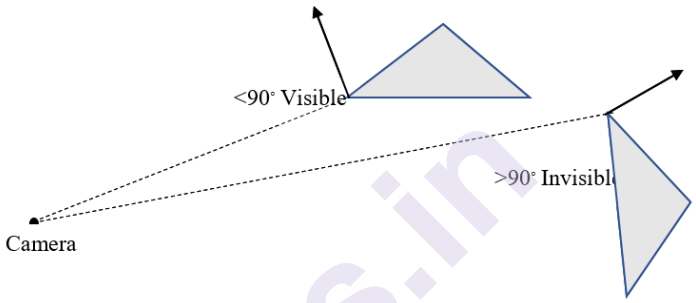


Fig:2.7 Polygon's Visibility

- Let's prove this concept algebraically. Let the camera be located at (0,0,0) and the polygon's vertex is (10, 10, 40). The normal vector is $[5\ 5\ -2]^T$

$$\mathbf{n}=\begin{bmatrix}\mathbf{5}\\ \mathbf{5}\\ \mathbf{-2}\end{bmatrix}$$

$\|\mathbf{n}\|=\sqrt{(5^2+5^2+(-2)^2)}=7.348$

The camera vector **c** is

$$\mathbf{c}=\begin{bmatrix}0-10\\ 0-10\\ 0-40\end{bmatrix}=\begin{bmatrix}-10\\ -10\\ -40\end{bmatrix}$$

$\|\mathbf{c}\| = \sqrt{((-10)^2 + (-10)^2 + (-40)^2)} = 42.426$

Therefore

$\|\mathbf{n}\| \cdot \|\mathbf{c}\| \cos(\beta)=5 \times (-10) + 5 \times (-10) + (-2) \times (-40)$

$7.348 \times 42.426 \times \cos(\beta) = -20$

$$\cos(\beta) =\frac{-20}{7.348 \times 42.426} = -0.0634$$

$\beta = \cos{-1}\ (-0.0634) = 93.635◦$

which shows that the polygon is invisible.

2.3.5.2 The Vector Product

- As mentioned above, there are two ways to obtain the product of two vectors.
- The first is the scalar product, and the second is the vector product, which is also called the cross product because of the '×' symbol used in its notation.
- It is based on the definition that two vectors **r** and **s** can be multiplied together to produce a third vector **t**:

r × s = t

where ||**t**|| = ||**r**|| · ||**s**||sin(β), and β is the angle between **r** and **s.**

The vector **t** is normal (90◦) to the plane containing the vectors **r** and **s**.

Once again, let's define two vectors and proceed to multiply them together:

r = ai + bj + ck

s = di + ej + fk

r × s = (ai + bj + ck) × (di + ej + fk) = ai × (di + ej + fk) + bj × (di + ej + fk) + ck ×(di + ej + fk)

r × s = ad (i × i) + ae (i × j) + af (i × k) + bd (j × i) +be j × j) +bf (j × k) + cd (k × i) + ce (k × j) +cf (k × k)

Using the definition for the cross product, operations such as (i×i), (j×j) and (k × k) result in a vector whose magnitude is 0. This is because the angle between the vectors is 0◦, and sin(0◦) = 0. Consequently, these terms disappear and we are left with

r × s = ae(i × j) + af(i × k) + bd(j × i) + bf(j × k) + cd(k × i) + ce(k × j)

The mathematician Sir William Rowan Hamilton assumed that i×j = k , j×k = i and k×i = j, but he also thought that j × i =- k, k × j =- i and i × k = -j.

Proceeding, then, with Hamilton's rules, we reduce the cross-product terms r x s to

r × s = ae(k) + af(−j) + bd(−k) + bf(i) + cd(j) + ce(−i) = (bf − ce)i + (cd − af)j + (ae − bd)k

We now modify the middle term to create a symmetric result:

r × s = (bf − ce)i − (af − cd)j + (ae − bd)k

$$\mathbf{r} \times \mathbf{s} = \begin{vmatrix} b & c \\ e & f \end{vmatrix}\mathbf{i} - \begin{vmatrix} a & c \\ d & f \end{vmatrix}\mathbf{j} + \begin{vmatrix} a & b \\ d & e \end{vmatrix}\mathbf{k}$$

Remember that **r × s** does not equal **s × r**.

2.3.5.2.1 The Right-Hand Rule

- The right-hand rule is an helper for working out the orientation of the cross-product vector.
- Given the operation r × s, if the right-hand thumb is aligned with **r**, the first finger with **s**, and the middle finger points in the direction of **t**.

2.4 DERIVING A UNIT NORMAL VECTOR FOR A TRIANGLE

- Figure 2.8 shows a triangle with vertices defined in an anti-clockwise sequence from its visible side. This is the side we want the surface normal to point upwards.

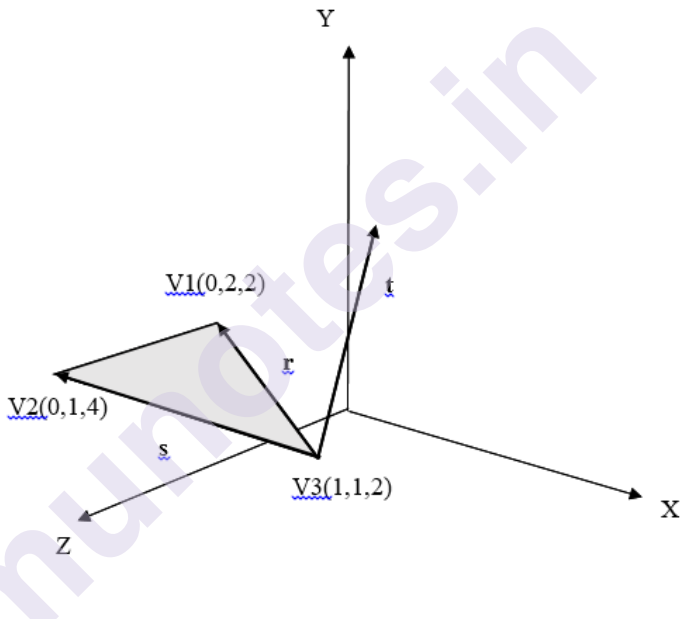


Fig:2.8 The normal vector t is derived from the cross-product r × s.

- Using the following information, we will compute the surface normal using the cross product and then convert it to a unit normal vector.
- Create vector **r** between v1 and v3, and vector **s** between v2 and v3:

$\mathbf{r} = -\mathbf{i} + \mathbf{j}$

$\mathbf{s} = -\mathbf{i} + 2\mathbf{k}$

$\mathbf{r} \times \mathbf{s} = \mathbf{t} = (1 \times 2 - 0 \times 0)\,\mathbf{i} - (-1 \times 2 - 0 \times -1)\,\mathbf{j} + (-1 \times 0 - 1 \times -1)\,\mathbf{k}$

$\mathbf{t} = 2\mathbf{i} + 2\mathbf{j} + \mathbf{k}$

$||\mathbf{t}|| = \sqrt{(2^2 + 2^2 + 1^2)} = 3$

$t_u = \frac{2}{3}\,\mathbf{i} + \frac{2}{3}\,\mathbf{j} + \frac{1}{3}\,\mathbf{k}$

The unit vector t_u can now be used in illumination calculations, and as it has unit length, dot product calculations are simplified.

2.5 AREAS

Figure 2.9 shows two 2D vectors, **r** and **s**. The height h =||s|| sin(β), therefore the area of the parallelogram is

$$||r||h = ||r|| \cdot ||s||\sin(\beta)$$

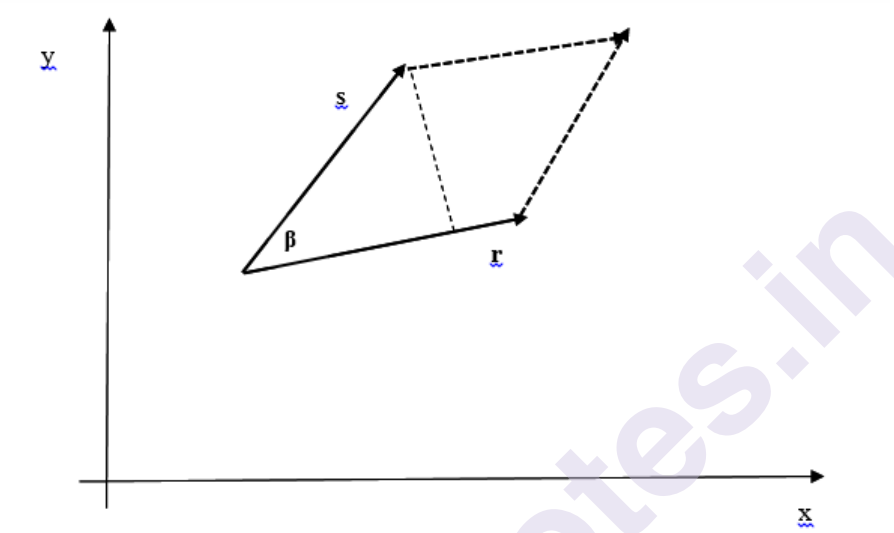


Fig:2.9 The area of the parallelogram formed by two vectors r and s equals ||r|| · ||s||sin β

But this is the magnitude of the cross product vector **t**.

Thus, when we calculate **r**×**s**, the length of the normal vector **t** equals the area of the parallelogram formed by **r** and **s**. Which means that the triangle formed by halving the parallelogram is half the area.

area of parallelogram = ||t||

area of triangle $=\frac{1}{2}$ ||t||

This means that it is a relatively easy exercise to calculate the surface area of an object constructed from triangles or parallelograms. In the case of a triangulated surface, we simply sum the magnitudes of the normal and halve the result.

2.5.1 Calculating 2D Areas

- Figure 2.10 shows three vertices of a triangle P_0(x0, y0), P_1(x1, y1) and P_2(x2, y2) formed in an anti-clockwise sequence. We can imagine that the triangle exists on the z = 0 plane, therefore the z-coordinates are zero.

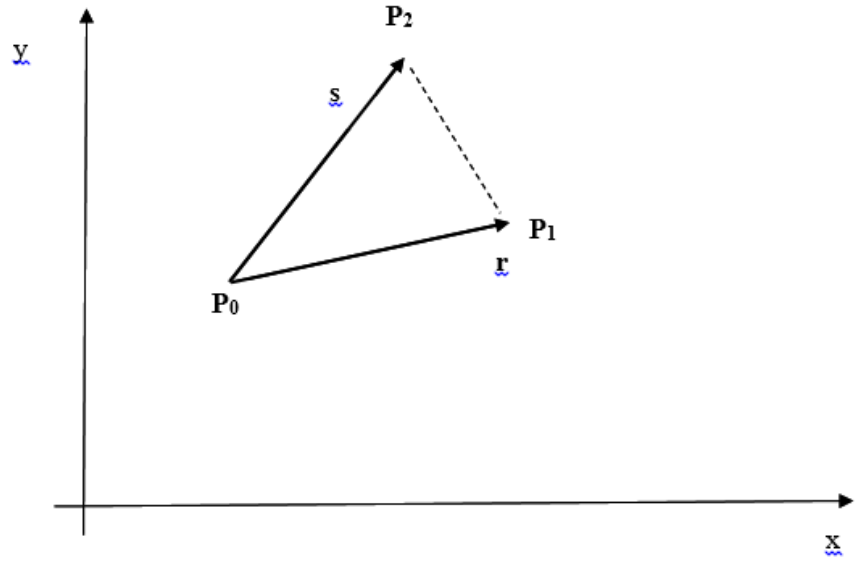


Fig:2.10 The area of the triangle formed by the vectors r and s is half the magnitude of their cross product.

The vectors **r** and **s** are computed as follows:

$$\mathbf{r} = \begin{bmatrix} x1 - x0 \\ y1 - y0 \\ 0 \end{bmatrix} \qquad \mathbf{s} = \begin{bmatrix} x2 - x0 \\ y2 - y0 \\ 0 \end{bmatrix}$$

$\mathbf{r} = (x1 - x0)i + (y1 - y0)j$

$\mathbf{s} = (x2 - x0)i + (y2 - y0)j$

$\|r \times s\| = (x1 - x0)(y2 - y0) - (x2 - x0)(y1 - y0)$

$= x1(y2 - y0) - x0(y2 - y0) - x2(y1 - y0) + x0(y1 - y0)$

$= x1y2 - x1y0 - x0y2 - x0y0 - x2y1 + x2y0 + x0y1 - x0y0$

$= x1y2 - x1y0 - x0y2 - x2y1 + x2y0 + x0y1$

$= (x0y1 - x1y0) + (x1y2 - x2y1) + (x2y0 - x0y2)$

But the area of the triangle formed by the three vertices is $\frac{1}{2}\|rxs\|$

Therefore

area $= \frac{1}{2}$ **[(x0y1 − x1y0) +(x1y2 − x2y1) + (x2y0 − x0y2)]**

2.6 SUMMARY:

Vectors are of fundamental importance in the study of 3D computer graphics, and we make extensive use of operations such as the dot product and cross product throughout the computer graphics.

2.7 QUESTIONS:

1) Explain in detail 3D vector manipulation.

2) Explain the following terms-

   a. Position Vectors

   b. Unit Vectors

   c. Cartesian Vectors

1) How Dot product helps in Back Face Detection?

   OR

   What is back face detection problem? State and explain how dot product is used to calculate back face detection.

2) Explain in detail Dot or Scalar product with suitable example.

3) How does Dot product help in Light Intensity calculation?

4) Applying the idea of dot product obtain the angle between two vectors given r (2,-3,4) and s (5,6,10).

5) Given a light source at (20,20,40) and the illuminated source as (0,10,0) and unit vector n (0,1,0) check the visibility of the object.

6) Explain how to drive a unit normal vector for a triangle.

2.8 REFERENCES:

Mathematics for Computer Graphics, John Vince, Springer-Verlag London,2nd Edition.

❄❄❄❄❄❄❄

3

TRANSFORMATION

Unit Structure :

3.0 OBJECTIVE:

In this chapter, we would investigate matrices as a tool for performing transformations

such as translations, rotations, and scales. We introduce the concept of four-dimensional homogeneous coordinates, which are widely used in 3D graphics systems to move between different coordinate spaces.

3.1 Introduction:

- Transformation means changing some graphics into something else by applying rules.
- In other words, we can define transformation as a change in object's properties.
- In computer graphics we can have various types of transformation such as translation, rotation, scaling etc.
- When a transformation takes place on 2D plane it is called as 2D transformation and when a transformation takes place on 3D plane it is called as 3D transformation.
- Transformation plays an important role in computer graphics to reposition the graphic on the screen or change their size or orientation.
- Although algebra is the basic notation for transformations, it is also possible to express them as matrices, which provide certain advantages for viewing the transformation and for interfacing to various types of computer graphics hardware.
- Transformations are used to scale, translate, rotate, reflect and shear shapes and objects.

3.2 2D TRANSFORMATION

When a transformation takes place on 2D plane it is called as 2D transformation.

3.2.1 Translation

- Translation is a type of transformation that moves an object to a different position on the screen.
- You can translate a point in 2D by adding translation coordinate(t_x,t_y) to the original coordinate (x,y) to get the new coordinate (x',y').
- t_x is a translation of an object about x-axis and t_y is a translation of an object about y-axis.
- Cartesian coordinates provide a one-to-one relationship between number and shape, such that when we change a shape's coordinates, we change its geometry.

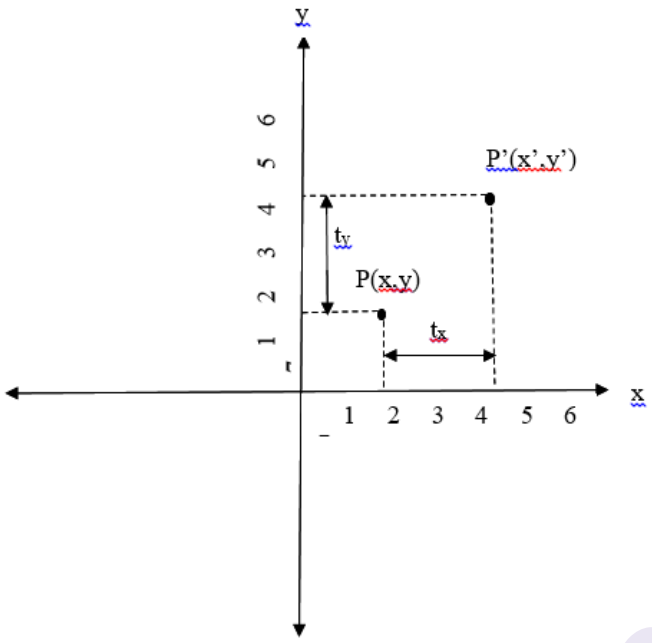


Fig:3.1 Translation of a point in 2D

From the above fig3.1 we can write:

$x' = x + t_x$

$y' = y + t_y$

t_x, t_y is called as translation vector or shift vector.

The above equation can be written in matrix form as:

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} x \\ y \end{bmatrix} + \begin{bmatrix} tx \\ ty \end{bmatrix}$$

3.2.2 Rotation

- In rotation we rotate the object at particular angle from its origin
- From the following figure 3.2 we can see that the point p(x ,y) is rotated is located at angle ϕ from horizontal x-axis and at distance r from the origin.

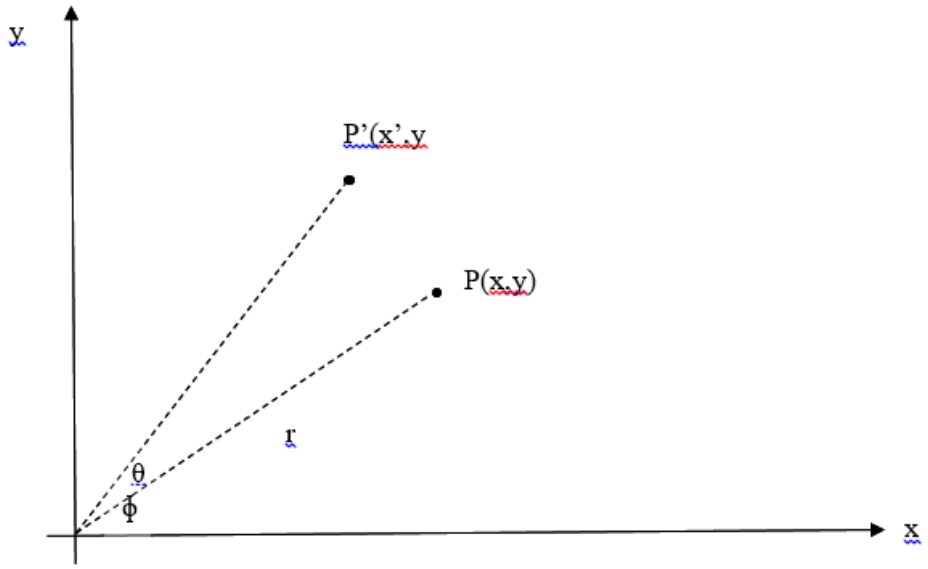


Fig:3.2 Rotation of a point in 2D

Let us suppose we want to rotate it at angle θ. After rotating a point p(x,y) we will get new point p'(x',y').

Using standard trigonometry the original coordinates of point p(x,y) can be represented as :

$x = r\cos\phi$ .....(1)

$y = r\sin\phi$. .....(2)

Same way we can represent the point p'(x',y') as −

$x' = r\cos(\phi+\theta) = r\cos\phi\cos\theta - r\sin\phi\sin\theta$ .....(3)

$y' = r\sin(\phi+\theta) = r\cos\phi\sin\theta + r\sin\phi\cos\theta$ .....(4)

substituting equation 1 and 2 in 3 and 4 we will get:

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

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

Representing the above equation in matrix form,

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

OR

p' = p . R

Where R is the rotation matrix

$$R=\begin{bmatrix}\cos\theta & \sin\theta\\ -\sin\theta & \cos\theta\end{bmatrix}$$

The rotation angle can be positive and negative.

For positive rotation angle, we can use the above rotation matrix. However, for negative angle of rotation, the matrix will change as shown below −

$$R=\begin{bmatrix} \cos(-\theta) & \sin(-\theta) \\ -\sin(-\theta) & \cos(-\theta) \end{bmatrix}$$

3.2.3 Scaling

- To change the size of an object, scaling transformation is used.
- In the scaling process, you either expand or compress the dimensions of the object.
- Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.

Let us assume that the original coordinates are X, Y, the scaling factors are (S_X, S_Y), and the produced coordinates are X′, Y′. This can be mathematically represented as shown below −

$X' = X \cdot S_X$ and $Y' = Y \cdot S_Y$

The scaling factor S_X, S_Y scales the object in X and Y direction respectively. The above equations can also be represented in matrix form as below –

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} X \\ Y \end{bmatrix} \cdot \begin{bmatrix} Sx & 0 \\ 0 & Sy \end{bmatrix}$$

Or

p’=p.S

Where S is the scaling matrix.

The scaling process is shown in the following figure 3.3.

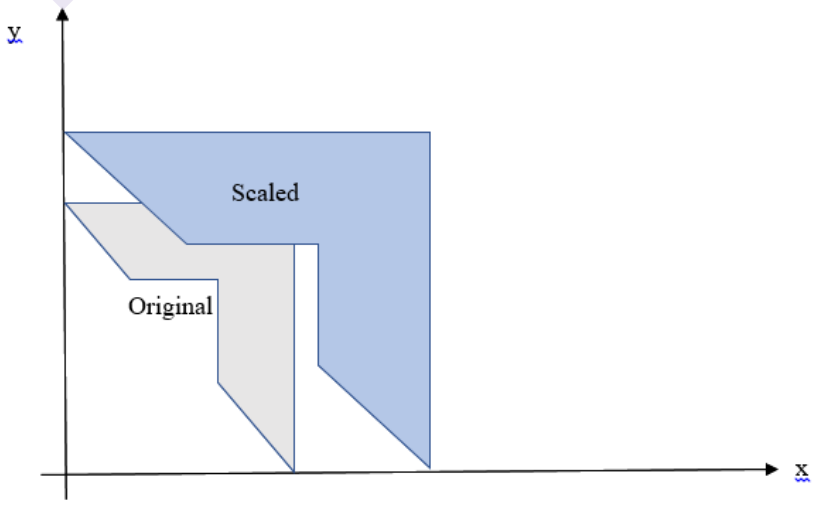


Fig:3.3 Scaling of an object in 2D

3.2.4 Reflection

- Reflection is the mirror image of original object.
- To make a reflection of a shape relative to the y-axis, we simply reverse the sign of the x -coordinate, leaving the y-coordinate unchanged

  $x = -x$ and $y = y$

- To reflect a shape relative to the x -axis we reverse the y-coordinates:

  $x = x$ and $y = -y$

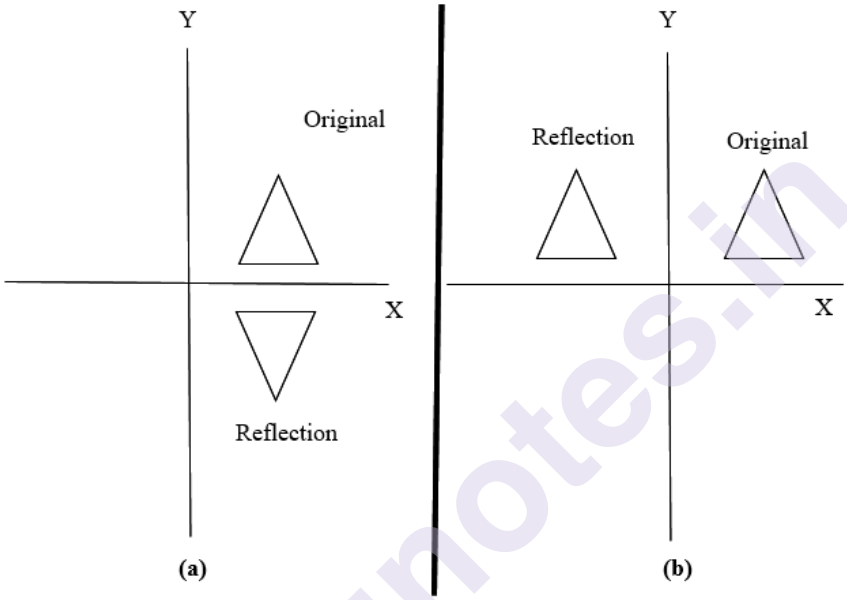


Fig:3.4 (a) A reflection of a shape relative to the x-axis (b) a reflection of a shape relative to the y-axis

3.2.5 Homogenous Coordinates

- To perform a sequence of transformation such as translation followed by rotation and scaling, we need to follow a sequential process –

  Translate the coordinates,

  Rotate the translated coordinates, and then

  Scale the rotated coordinates to complete the composite transformation.

- To shorten this process, we have to use 3×3 transformation matrix instead of 2×2 transformation matrix.
- To convert a 2×2 matrix to 3×3 matrix, we have to add an extra dummy coordinate W.
- In this way, we can represent the point by 3 numbers instead of 2 numbers, which is called Homogenous Coordinate system.

- In this system, we can represent all the transformation equations in matrix multiplication.
- Following are matrix for two-dimensional transformation in homogeneous coordinate:

Translation T= $\begin{bmatrix} 1 & 0 & tx \\ 0 & 1 & ty \\ 0 & 0 & 1 \end{bmatrix}$ **OR** $= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ tx & ty & 1 \end{bmatrix}$

Rotation (Clockwise) R= $\begin{bmatrix} Cos\theta & -sin\theta & 0 \\ sin\theta & Cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Rotation (Anticlockwise) R= $\begin{bmatrix} Cos\theta & sin\theta & 0 \\ -sin\theta & Cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Scaling S= $\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Reflection against x-axis= $\begin{bmatrix} 1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Reflection against y-axis= $\begin{bmatrix} -1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

3.3 MATRICES

- Matrix notation was investigated by the British mathematician **Arthur Cayley** around **1858**.
- Caley formalized matrix algebra, along with the American mathematicians Benjamin and Charles Pierce.
- Also, by the start of the 19th century Carl Gauss (1777–1855) had proved that transformations were not commutative, i.e. T1 × T2 ≠ T2 × T1, and Caley's matrix notation would clarify such observations.
- For example, consider the transformation T1:

$$\mathbf{T}_1 \quad \begin{aligned} x' &= ax + by \\ y' &= cx + dy \end{aligned}$$

and another transformation T2 that transforms T1:

$$\mathbf{T}_2 \times \mathbf{T}_1 \quad \begin{aligned} x'' &= Ax' + By' \\ y'' &= Cx' + Dy' \end{aligned}$$

If we substitute the full definition of T1 we get

$$\mathbf{T}_2 \times \mathbf{T}_1 \quad \begin{aligned} x'' &= A(ax + by) + B(cx + dy) \\ y'' &= C(ax + by) + D(cx + dy) \end{aligned}$$

which simplifies to

$$\mathbf{T}_2 \times \mathbf{T}_1 \quad \begin{aligned} x'' &= (Aa + Bc)x + (Ab + Bd)y \\ y'' &= (Ca + Dc)x + (Cb + Dd)y \end{aligned}$$

Caley proposed separating the constants from the variables, as follows:

$$\mathbf{T}_1 \begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

where the square matrix of constants in the middle determines the transformation.

$$x' = ax + by$$

$$y' = cx + dy$$

Using Caley's notation, the product T2 × T1 is

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} x' \\ y' \end{bmatrix}$$

But the notation also intimated that

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

and when we multiply the two *inner matrices* together they must produce

$$x'' = (Aa + Bc)x + (Ab + Bd)y$$

$$y'' = (Ca + Dc)x + (Cb + Dd)y$$

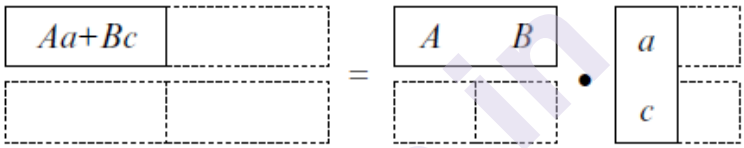
or in matrix form

$$\begin{bmatrix} x'' \\ y'' \end{bmatrix} = \begin{bmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix}$$

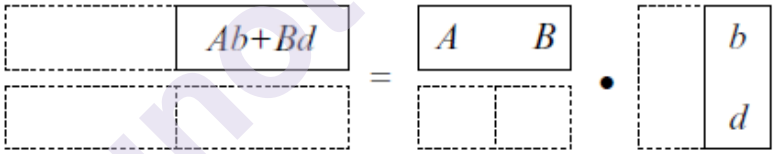
otherwise, the two systems of notation will be inconsistent. This implies that

$$\begin{bmatrix} Aa + Bc & Ab + Bd \\ Ca + Dc & Cb + Dd \end{bmatrix} = \begin{bmatrix} A & B \\ C & D \end{bmatrix} \cdot \begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

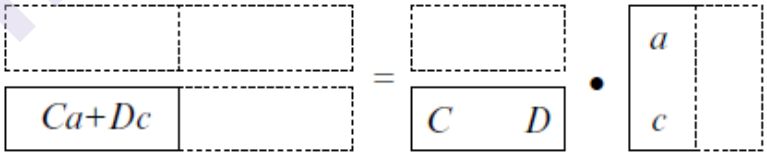
which demonstrates how matrices must be multiplied. Here are the rules for matrix multiplication:



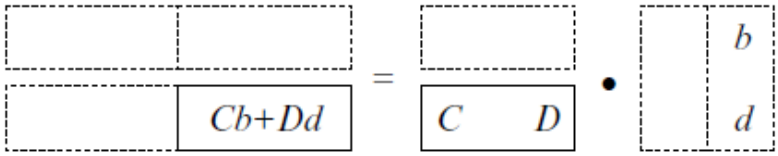
1. The top left-hand corner element *Aa+Bc* is the product of the top row of the first matrix by the left column of the second matrix.



2. The top right-hand element *Ab + Bd* is the product of the top row of the first matrix by the right column of the second matrix.



3. The bottom left-hand element *Ca + Dc* is the product of the bottom row of the first matrix by the left column of the second matrix.



4. The bottom right-hand element *Cb+Dd* is the product of the bottom row of the first matrix by the right column of the second matrix.

It is now a trivial exercise to confirm Gauss's observation that T1 × T2 ≠ T2 × T1, because if we reverse the transforms T2 × T1 to T1 × T2 we get

$$\begin{bmatrix} Aa+Bc & Ab+Bd \\ Ca+Dc & Cb+Dd \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} A & B \\ C & D \end{bmatrix}$$

which shows conclusively that the product of two transforms is not commutative.

3.3.1 Determinant of a Matrix

The *determinant* of a 2×2 matrix is a scalar quantity computed. Given a matrix

$$\begin{bmatrix} a & b \\ c & d \end{bmatrix}$$

its determinant is $ad - cb$ and is represented by

$$\begin{vmatrix} a & b \\ c & d \end{vmatrix}$$

For example, the determinant of $\begin{bmatrix} 3 & 2 \\ 1 & 2 \end{bmatrix}$ is $3 \times 2 - 1 \times 2 = 4$

3.4 3D TRANSFORMATION:

When a transformation of an object takes place in 3D Plane that it is known as 3D Transformation.

3.4.1 Rotation in 3D

- 3D rotation is not same as 2D rotation.
- In 3D rotation, we have to specify the angle of rotation along with the axis of rotation.
- We can perform 3D rotation about X, Y, and Z axes.
- They are represented in the matrix form as below –

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

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

$$R_z(\theta)=\begin{matrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$$

- The above rotations are also known as **yaw, pitch and roll**.
- The roll, pitch and yaw angles can be defined as follows:

  roll is the angle of rotation about the **z -axis**

  pitch is the angle of rotation about the **x -axis**

  yaw is the angle of rotation about the **y-axis**

3.4.2 Translation in 3D

- In Computer graphics,3D Translation is a process of moving an object from one position to another in a three-dimensional plane.
- The process of translation in 3D is similar to 2D translation.
- A point can be translated in 3D by adding translation coordinates (t_x,t_y,t_z) to the original coordinates (x,y,z) to get the new coordinates (x',y',z').
- Translation matrix is given by:-

$$T==\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{bmatrix} \textbf{ or } \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.3 Scaling in 3D

- You can change the size of an object using scaling transformation.
- In the scaling process, you either expand or compress the dimensions of the object.
- Scaling can be achieved by multiplying the original coordinates of the object with the scaling factor to get the desired result.
- In 3D scaling operation, three coordinates are used.
- Let us assume that the original coordinates are X,Y,Z scaling factors are (SX,SY,Sz) respectively, and the produced coordinates are X′,Y′,Z′.
- This can be mathematically represented as shown below –

$$S=\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

It can be written as

$$[x' \quad y' \quad z'] = [x \quad y \quad z] \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

3.4.4 Homogenous Transformation Matrices for 3D

Transformation matrices is a basic tool for transformation usually 3x3 or 4x4 matrix are used for transformation. The following are the homogenous matrices for various operation:

Translation T = $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ tx & ty & tz & 1 \end{bmatrix}$ **or** $\begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Scaling S = $\begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$

Rotation about x axis $R_x(\theta)$= $\begin{matrix} 1 & 0 & 0 & 0 \\ 0 & cos\theta & -sin\theta & 0 \\ 0 & sin\theta & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$

Rotation about y-axis Ry(θ) = $\begin{matrix} cos\theta & 0 & sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -sin\theta & 0 & cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$

Rotation about $R_z(\theta)$ = $\begin{matrix} cos\theta & -sin\theta & 0 & 0 \\ sin\theta & Cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{matrix}$

3.5 2D ROTATION ABOUT AN ARBITRARY POINT

A rotation about the origin is given by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Therefore, using matrices, we can develop a rotation about an arbitrary point (px, py) as follows:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = [\text{translate}(px, py)] \cdot [\text{rotate } \theta] \cdot [\text{translate}(-px, -py)] \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Which expands to

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = = \begin{bmatrix} 1 & 0 & px \\ 0 & 1 & py \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -px \\ 0 & 1 & -py \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

We can now concatenate these matrices into a single matrix by multiplying them together. Let's begin by multiplying the rotate θ and the translate (−px, −py) matrices together. This produces

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & px \\ 0 & 1 & py \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & -\text{px}\cos(\theta) + \text{py}\sin(\theta) \\ \sin\theta & \cos\theta & -\text{px}\sin(\theta) - \text{py}\cos(\theta) \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

and finally we will get :-

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta & \text{px}(1 - \cos(\theta)) + \text{py}\sin(\theta) \\ \sin\theta & \cos\theta & \text{py}(1 - \cos(\theta)) - \text{px}\sin(\theta) \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

Above is the matrix for 2D Rotation about an Arbitrary Point.

3.6 CHANGE OF AXES

- Points in one coordinate system often have to be referenced in another one.
- For example, to view a 3D scene from an arbitrary position, a virtual camera is positioned in the world space using a series of transformations. An object's coordinates, which are relative to the world frame of reference, are computed relative to the camera's axial system, and then used to develop a perspective projection.

3.6.1 2D Change of Axes

- Figure 3.5 shows a point p(x, y) relative to the XY -axes, but we require to know the coordinates relative to the X'Y' -axes.

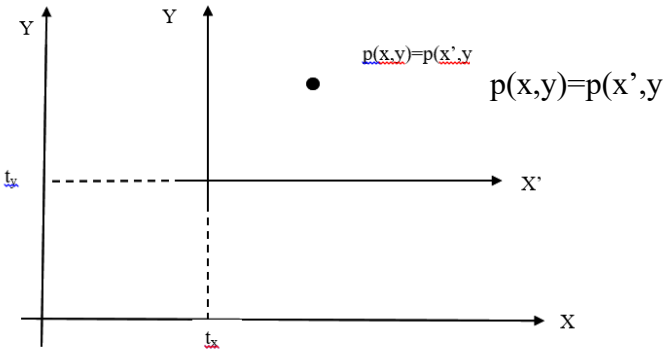


Fig. 3.5 The X Y -axes are translated by (tx, ty).

- To do this, we need to know the relationship between the two coordinate systems, and ideally, we want to apply a technique that works in 2D and 3D.

- If the second coordinate system is a simple translation (tx, ty) relative to the reference system, as shown in Figure 3.5, the point p(x, y) has coordinates relative to the translated system (x − tx, y − ty):

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -tx \\ 0 & 1 & -ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

If the X' Y' -axes are rotated θ relative to the XY -axes, as shown in Figure 3.6, a point P(x, y) relative to the XY -axes has coordinates (x' , y') relative to the rotated axes given by

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

which simplifies to

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

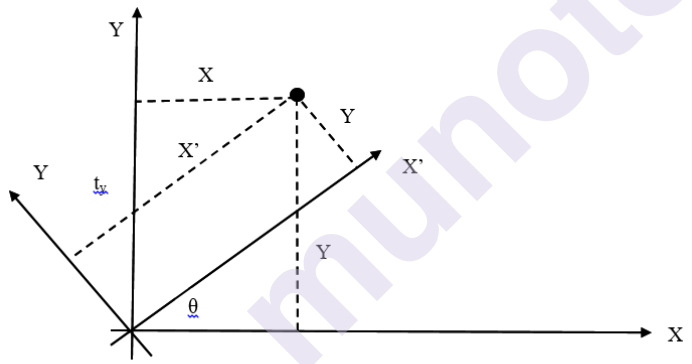


Fig. 3.6 The secondary set of axes are rotated by θ.

- When a coordinate system is rotated and translated relative to the reference system, a point p (x, y) has coordinates (x' , y') relative to the new axes given by

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & 0 \\ -\sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & -tx \\ 0 & 1 & -ty \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

which simplifies to

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & \sin(\theta) & -\text{tx}\cos(\theta) - \text{ty}\sin(\theta) \\ -\sin(\theta) & \cos(\theta) & \text{tx}\sin(\theta) - \text{ty}\cos(\theta) \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

3.7 DIRECTION COSINES

- Direction cosines are the cosines of the angles between a vector and the axes, and for unit vectors they are the vector's components.
- Figure 3.7 shows two unit vectors X' and Y' , and by inspection the direction cosines for X' are cos(β) and cos(90◦ −β), which can be rewritten as cos(β) and sin(β), and the direction cosines for Y' cos(90◦ + β) and cos(β), which can be rewritten as − sin(β) and cos(β).

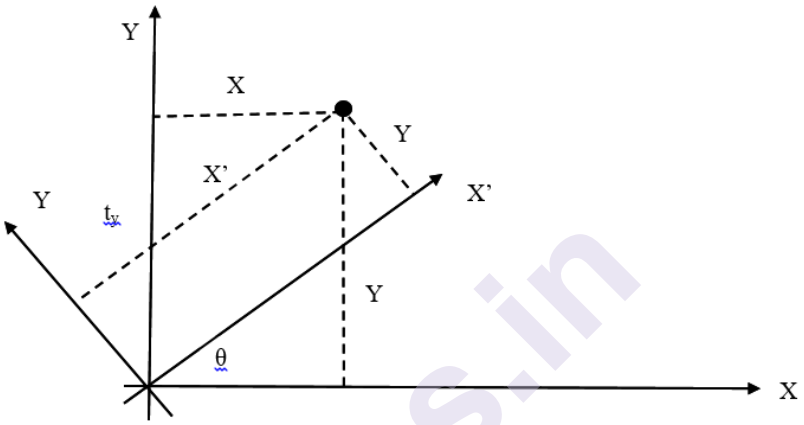


Fig. 3.6 The secondary set of axes are rotated by θ.

- But these direction cosines cos(β), sin(β), − sin(β) and cos(β) are the four elements of the rotation matrix used above

$$\begin{bmatrix} \cos\beta & \sin\beta \\ -\sin\beta & \cos\beta \end{bmatrix}$$

- The top row contains the direction cosines for the X' -axis and the bottom row contains the direction cosines for the Y' -axis.
- This relationship also holds in 3D.

3.8 TRANSFORMING VECTORS

- The transforms described in this chapter have been used to transform single points. However, a geometric database will contain not only pure vertices, but also vectors, which must also be subject to any usual transform.
- A generic transform Q of a 3D point can be represented by

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [Q] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

and as a vector is defined by two points we can write

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [Q] \begin{bmatrix} x2 - x1 \\ y2 - y1 \\ z2 - z1 \\ 1 - 1 \end{bmatrix}$$

where we see the homogeneous scaling term collapse to zero. This implies that any vector $[x \quad y \quad z]^T$ can be transformed using

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = [Q] \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

3.9 PERSPECTIVE PROJECTION

- Of all the projections employed in computer graphics, the perspective projection is the one most widely used.
- There are two stages to its computation: the first stage involves converting world coordinates to the camera's frame of reference, and the second stage transforms camera coordinates to the projection plane coordinates.
- We have already looked at the transforms for locating a camera in world space, and the inverse transform for converting world coordinates to the camera's frame of reference.
- Let's now investigate how these camera coordinates are transformed into a perspective projection.

We begin by assuming that the camera is directed along the z -axis as shown in Figure 3.8. Positioned d units along the axis is a projection screen, which will be used to capture a perspective projection of an object.

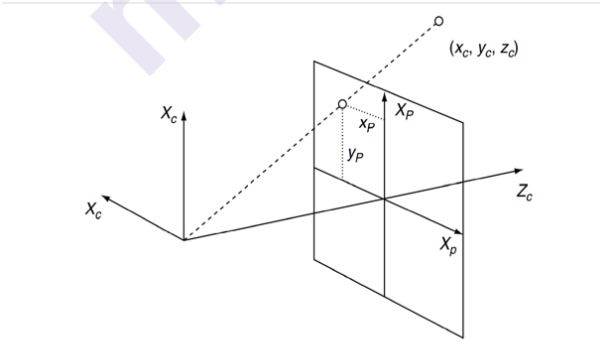


Fig 3.8 The axial systems used to produce a perspective projection.

- Figure 3.8 shows that any point (xc, yc, zc) becomes transformed to (xs, ys, d). It also shows that the screen's x -axis is pointing in the opposite direction to the camera's x -axis, which can be compensated for by reversing the sign of xs when it is computed.

- Figure 3.9 shows plan and side views of the scenario depicted in Figure 3.8, which enables us to inspect the geometry and make the following observations:

$$\frac{x}{z} = \frac{-xp}{d} \qquad xp = -d\,\frac{x}{z} \qquad xp = \frac{-y}{z/d}$$

$$\frac{y}{z} = \frac{yp}{d} \qquad yp = d\frac{y}{z} \qquad yp = \frac{y}{z/d}$$

This can be expressed in matrix as

$$\begin{bmatrix} xs \\ ys \\ zs \\ W \end{bmatrix} = \begin{bmatrix} -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

If we multiply it out we get,

$$[xp \quad yp \quad zp \quad W]^{T} = [-x \quad y \quad z \quad z/d]^{T}$$

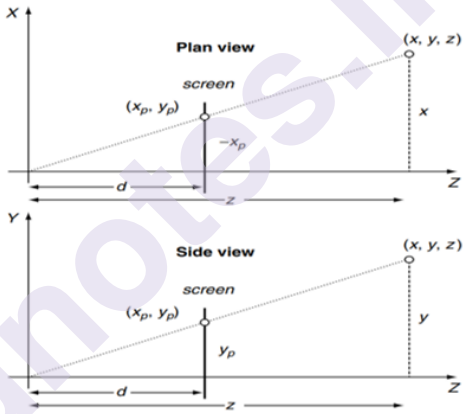


Fig:3.9: The plan and side views for computing the perspective projection transform.

- The idea behind homogeneous coordinates says that we must divide the terms xp, yp, zp by W to get the scaled terms, which produces the following:-

$$xp = \frac{-x}{z/d} \qquad yp = \frac{y}{z/d} \qquad zp = \frac{z}{z/d} = d$$

3.10 SUMMARY:

- Transformation means change in object's property. It can be 2D or 3D transformation.
- Translation will move the object to the new position, rotation will rotate the object by some angle of degree, and scaling will expand or collapse the object
- Homogeneous Coordinate matrix are used to make calculation easy.

3.11 QUESTION:

1) Explain 3D translation, 3D Scaling with suitable examples.

2) Write a short note on 3D rotation.

3) Write a short note on 2D transformations.

4) What is 3D transformation? State and explain scaling and translation in 3D.

5) What is transformation? State and explain the concept of translation in 2D and 3D.

6) Write a short note on 2D rotation.

7) Explain the concept of perspective projection.

8) Explain the concept of direction cosine.

9) Write a short note on Change of axis.

10) Give homogeneous coordinate matrix for various transformation operation in 3D.

3.12 REFERENCES:

Mathematics for Computer Graphics, John Vince, Springer-Verlag London,2nd Edition.

❄❄❄❄❄❄❄

4

GRAPHICS PROCESSING UNIT

Unit Structure :

4.0 OBJECTIVES:

This chapter would make you understand the following concept:

- What is DirectX?
- Differences between GPU and CPU.
- How GPU works?

4.1 INTRODUCTION TO DIRECTX

- Microsoft DirectX is a collection of application programming interfaces (APIs) for handling tasks related to multimedia, especially game programming and video, on Microsoft platforms.
- DirectX is a series of application programming interfaces (API) that provide low-level access to hardware components like video cards, the sound card, and memory. At a basic level, DirectX allows games to "talk" to video cards.
- In the DOS days, games had direct access to video cards and the motherboard, and you could directly edit the configuration file to make changes.
- But with Windows 95, Microsoft restricted access to low-level hardware as a security measure.
- That meant that games could no longer interact with low-level hardware features, and it was a problem. So, to facilitate that access,

Microsoft introduced DirectX — think of DirectX as a middleman that facilitates communication between a game and a video card.

- Originally, the names of these APIs all began with "Direct", such as Direct3D, DirectDraw, DirectMusic, DirectPlay, DirectSound, and so on. 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.
- DirectX lets developers unlock the full potential of your computer's hardware.

4.2 UNDERSTANDING GPU (GRAPHICS PROCESSING UNIT)

- A graphics processing unit (**GPU**) is a computer chip that renders graphics and images by performing rapid mathematical calculations.
- GPUs are used for both professional and personal computing.
- Traditionally, GPUs are responsible for the rendering of 2D and 3D images, animations and video. Even now, they have a wider use range.
- In the early days of computing, the central processing unit (**CPU**) performed these calculations. As more graphics-intensive applications were developed, however, their demands put a stress on the CPU and decreased performance.
- GPUs were developed as a way to unload those tasks from CPUs and to improve the rendering of 3D graphics.
- GPUs work by using a method called parallel processing, where multiple processors handle separate parts of the same task.
- GPUs are well known in PC (personal computer) gaming, allowing for smooth, high-quality graphics rendering. Developers also began using GPUs as a way to accelerate workloads in areas such as artificial intelligence (AI).

Some examples of GPU use cases include:

- GPUs can accelerate the rendering of real-time 2D and 3D graphics applications.
- Video editing and creation of video content has improved with GPUs. Video editors and graphic designers, for example, can use the parallel processing of a GPU to make the rendering of high-definition video and graphics faster.
- Video game graphics have become more intensive computationally, so in order to keep up with display technologies -- like 4K and high refresh rates -- emphasis has been put on high-performing GPUs.

- GPUs can accelerate machine learning. With the high-computational ability of a GPU, workloads such as image recognition can be improved.
- GPUs can share the work of CPUs and train deep learning neural networks for AI applications. Each node in a neural network performs calculations as part of an analytical model. Programmers eventually realized that they could use the power of GPUs to increase the performance of models across a deep learning matrix -- taking advantage of far more parallelism than is possible with conventional CPUs.

4.3 HOW GPU WORKS

- A GPU may be found integrated with a CPU on the same electronic circuit, on a graphics card or in the motherboard of a personal computer or server.
- GPUs and CPUs are fairly similar in construction.
- However, GPUs are specifically designed for performing more complex mathematical and geometric calculations. These calculations are necessary to render graphics.
- GPUs may contain more transistors than a CPU.
- GPUs will use parallel processing, where multiple processors handle separate parts of the same task. A GPU will also have its own RAM (random access memory) to store data on the images it processes.
- Information about each pixel is stored, including its location on the display.
- A digital-to-analog converter (DAC) is connected to the RAM and will turn the image into an analog signal so the monitor can display it. Video RAM will typically operate at high speeds.
- GPUs will come in two types: **integrated and discrete**. Integrated GPUs come embedded alongside the GPU, while discrete GPUs can be mounted on a separate circuit board.
- For companies that require heavy computing power, or work with machine learning or 3D visualizations, having GPUs fixated in the cloud may be a good option. An example of this is Google's Cloud GPUs, which offer high-performance GPUs on Google Cloud. Hosting GPUs in the cloud will have the benefits of freeing up local resources, saving time, cost and scalability.
- Users can choose between a range of GPU types while gaining flexible performance based on their needs.

4.4 GPU VS. CPU

- GPUs are fairly similar to CPU architectures. However, CPUs are used to respond to and process the basic instructions that drive a computer, while GPUs are designed specifically to quickly render high-resolution images and video. Essentially, CPUs are responsible for interpreting most of a computer's commands, while GPUs focus on graphics rendering.

- In general, a GPU is designed for data-parallelism and applying the same instruction to multiple data-items (SIMD). A CPU is designed for task-parallelism and doing different operations.

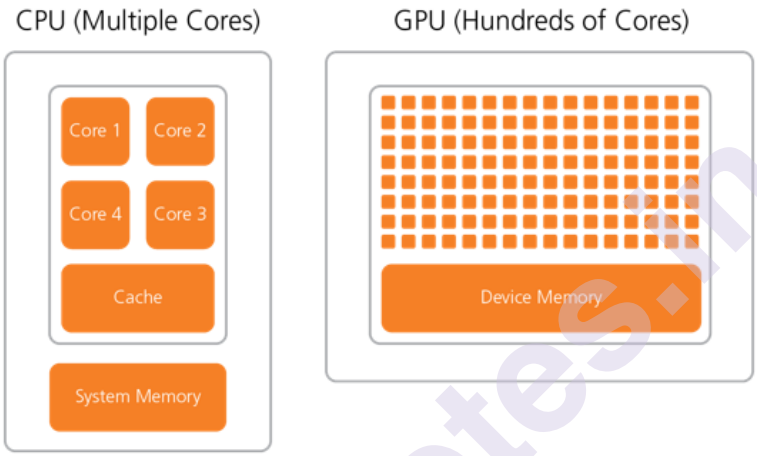


Fig: 4.1 CPU V/s GPU

- Both are also differentiated by the number of cores. The core is essentially the processor within the processor. Most CPU cores are numbered between four and eight, though some have up to 32 cores. Each core can process its own tasks, or threads. Because some processors have multithreading capability in which the core is divided virtually, allowing a single core to process two threads -- the number of threads can be much higher than the number of cores. This can be useful in video editing and transcoding. CPUs can run two threads (independent instructions) per core (the independent processor unit). A GPU core can have four to 10 threads per core.

- A GPU is able to render images more quickly than a CPU because of its parallel-processing architecture, which allows it to perform multiple calculations at the same time. A single CPU does not have this capability, although multicore processors can perform calculations in parallel by combining more than one CPU onto the same chip.

- A CPU also has a higher clock speed, meaning it can perform an individual calculation faster than a GPU, so it is often better equipped to handle basic computing tasks.

4.5 GPU ARCHITECTURE

- If we inspect the high-level architecture overview of a GPU (again, strongly depended on make/model), it looks like the nature of a GPU is all about putting available cores to work and it's less focused on low latency cache memory access.

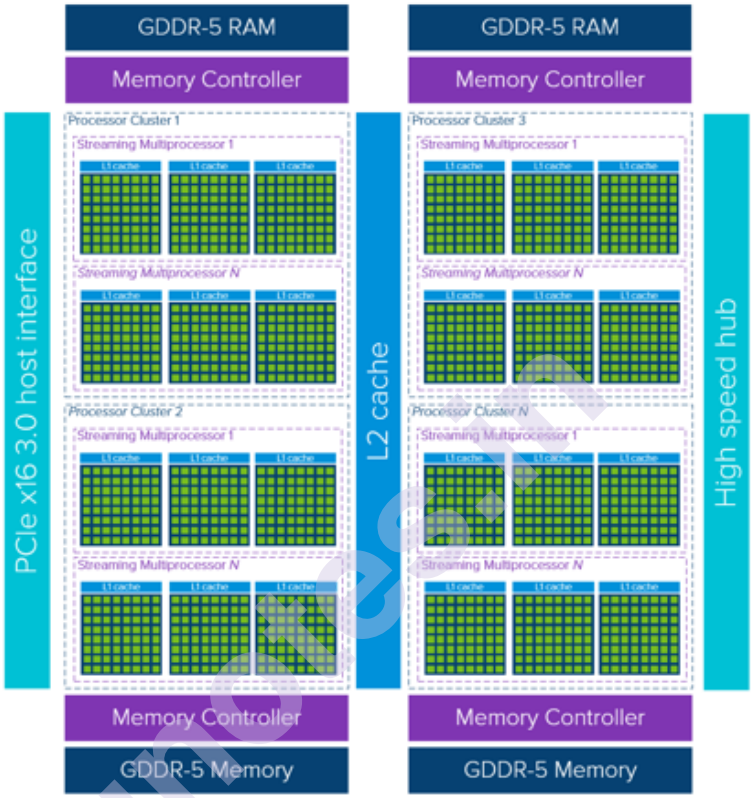


Fig:4.2 GPU Architecture

- A single GPU device consists of multiple Processor Clusters (PC) that contain multiple Streaming Multiprocessors (SM).
- Each SM accommodates a layer-1 instruction cache layer with its associated cores.
- Typically, one SM uses a dedicated layer-1 cache and a shared layer-2 cache before pulling data from global GDDR-5 memory.
- Its architecture is tolerant of memory latency.
- Compared to a CPU, a GPU works with fewer, and relatively small, memory cache layers.
- Reason being is that a GPU has more transistors dedicated to computation meaning it cares less how long it takes the retrieve data from memory.
- The potential memory access 'latency' is masked as long as the GPU has enough computations at hand, keeping it busy.

- A GPU is optimized for data parallel throughput computations.
- Looking at the numbers of cores it quickly shows you the possibilities on parallelism that is it is capable of.
- When examining the current NVIDIA flagship offering, the Tesla V100, one device contains 80 SM's, each containing 64 cores making a total of 5120 cores! Tasks aren't scheduled to individual cores, but to processor clusters and SM's.
- That's how it's able to process in parallel. Now combine this powerful hardware device with a programming framework so applications can fully utilize the computing power of a GPU.

In 2020, some of the top GPUs and graphics cards have included:

GeForce RTX 3080

GeForce RTX 3090

GeForce RTX 3060 Ti

AMD Radeon RX 6800 XT

AMD Radeon RX 5600 XT

4.6 SUMMARY:

- Modern GPUs are very efficient at manipulating computer graphics and image processing, and their highly parallel structure makes them more efficient than general-purpose CPUs for algorithms where the processing of large blocks of data is done in parallel.
- The CPU (central processing unit) has often been called the brains of the PC. But increasingly, that brain is being enhanced by another part of the PC – the GPU (graphics processing unit), which is its soul.
- GPU are very useful for rendering 2D and 3D Graphics.

4.7 QUESTIONS:

1) What is DirectX?
2) Write a note on GPU.
3) What is the difference between the CPU and GPU?
4) Explain in detail GPU architecture.
5) Explain how GPU works.

4.8 REFERENCES:

https://searchvirtualdesktop.techtarget.com/

https://blogs.vmware.com/

❄❄❄❄❄❄❄

5

DIRECTX 11

Unit Structure :

5.0 OBJECTIVES:

1. To obtain basic understanding of Direct3D's role in programming 3D H/W.
2. To understand the role of COM.
3. To learn fundamentals of Graphics concepts.
4. To understand how to initialize Direct3D.

5.1 OVERVIEW:

Graphics API Direct3D is used to render 3D scenes with 3D hardware acceleration. There are various software interfaces that are provided by Direct3D to control hardware for example, to instruct the graphics hardware to clear the render target, like the screen, method like **ID3D11DeviceContext::ClearRenderTargetView** is used.

For any Direct3D 11 capable device Direct3D plays important role as an interface between software and graphics hardware. A Direct3D 11 capable graphics device must support the entire Direct3D 11 capability set. In the case of Direct3D 9, a device only had to support a subset of Direct3D 9 capabilities. In Direct3D 11 device capability checking is not required because it is mandatory to implement entire capability list.

5.2 COM

Component Object Model (COM) technology allows DirectX to be independent of any programming language and provides backwards compatibility.

COM is used as a C++ class and referred as an interface. Details are usually hidden from programmers when using COM.

We obtain pointers to COM interfaces through some special functions of another COM interface; C++ **new** keyword is not used as we generally do. **Release** method is called after we are done with any COM interface and it performs memory management.

COM interfaces are prefixed with a capital I. For example, the COM interface that represents a 2D texture is called **ID3D11Texture2D**.

5.3 TEXTURES AND RESOURCES FORMATS

Textures are used for creating image data. A 2D texture is a matrix of data elements. In an image a texture stores pixel colors of that image.

However, in an advanced technique called normal mapping, each element in the texture stores a 3D vector instead of a color. Textures are more general purpose than just storing image data. A 1D texture is equivalent to a 1D array of data elements, and a 3D texture is as a 3D array of data elements.

Textures are more than just arrays of data; they can have mipmap levels, and the GPU operations on them, as applying filters and multisampling. A texture can only store certain kinds of data formats, which are described by the **DXGI_FORMAT** enumerated type.

Some example formats are:

DXGI_FORMAT_R32G32B32_FLOAT	Every element has three 32-bit floating-point components.
DXGI_FORMAT_R16G16B16A16_UNORM	Every element has four 16-bit components mapped to the [0, 1] range.
DXGI_FORMAT_R32G32_UINT	Every element has two 32-bit unsigned integer components.
DXGI_FORMAT_R8G8B8A8_UNORM	Every element has four 8-bit unsigned components mapped to the [0, 1] range.
DXGI_FORMAT_R8G8B8A8_SNORM	Every element has four 8-bit signed components mapped to the [−1, 1] range.

DXGI_FORMAT_R8G8B8A8_SINT	Every element has four 8-bit signed integer components mapped to the [−128, 127] range.
DXGI_FORMAT_R8G8B8A8_UINT	Every element has four 8-bit unsigned integer components mapped to the [0, 255] range.

Here R, G, B, A letters are used to stand for red, green, blue, and alpha, respectively. Colors are created by combinations of R, G, B.

The format

DXGI_FORMAT_R32G32B32_FLOAT

has three floating-point components and can therefore store a 3D vector with floating-point coordinates.

The *typeless* formats are also present, where we just reserve memory and then specify how to reinterpret the data at a later; for example, the following typeless format reserves elements with four 8-bit components, but does not have any specific data type :

DXGI_FORMAT_R8G8B8A8_TYPELESS

5.4 THE SWAP CHAIN AND PAGE FLIPPING

Flickering is one of the problem we may face when drawing a scene on screen, to avoid this, we use a screen texture known as back buffer. To make sure the end user will see the entire scene or animation on screen, first we draw it on the back buffer and after the completion, it will be passed to the screen.

For implementation, we require another buffer known as front buffer, which stores the display data to be drawn currently on the monitor and the next scene/frame of animation is drawn on the back buffer.

The role of back and front buffers will be reversed after the frame is completely drawn on the screen: hence, for the next frame back buffer becomes front buffer and front becomes back buffer.

This swapping mechanism is also known as *presenting*. This presenting actually swaps the pointers of back and front buffer. Figure 5.1 illustrates the process.

This continuous operation of swapping the two buffers forms a *swap chain*. To represent this swap chain the **IDXGISwapChain** interface is used. This interface is used for storing the textures of front and back buffers and provides methods for resizing and presenting (**IDXGISwapChain::ResizeBuffers**, **IDXGISwapChain::Present**). Using two buffers for this purpose is known as double buffering.

Note that even though the back buffer is a texture (so an element should be called a texel), we often call an element a pixel because, in the case of the back buffer, it stores color information.

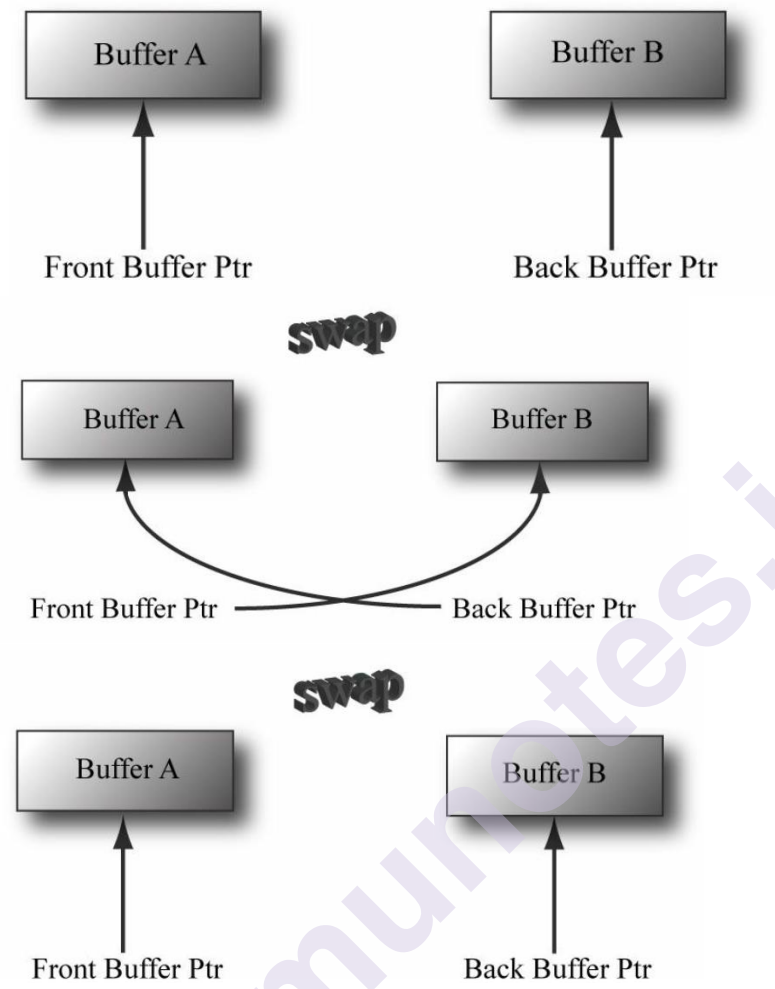


Figure 5.1. From top-to-bottom, we first render to Buffer B, which is serving as the current back buffer. Once the frame is completed, the pointers are swapped and Buffer B becomes the front buffer and Buffer A becomes the new back buffer. We then render the next frame to Buffer A. Once the frame is completed, the pointers are swapped and Buffer A becomes the front buffer and Buffer B becomes the back buffer again.

5.5 DEPTH BUFFERING

The third buffer we use not to store image data but to store the depth of the particuar pixel. Depth values range from 0.0 to 1.0, where 0.0 means the object is closest to the viewer and 1.0 means the object is farther from the viewer.

The pixel value is back buffer and the depth buffer has one to one correspondence i.e *ij*th element in back buffer corresponds to *ij*th element in the depth buffer. So we can say that the number of pixel we have in back buffer are same as the entries we will have in depth buffer.

Figure 5.2 shows a simple scene, where some objects partially obscure the objects behind them. In order for Direct3D to determine which pixels of an object are in front of another, it uses a technique called depth buffering or z-buffering. Let us emphasize that with depth buffering, the order in which we draw the objects does not matter.

Consider the example given in Figure 5.3 to understand the concept of a depth buffer, the example shows the volume the viewer sees and a 2D side view of that volume. From the figure, we see three different pixels are competing to be rendered onto the pixel P position on the view window. We as humans know that the closest pixel will be drawn as position P but computer doesn't. Before the rendering starts, the back buffer will be cleared to one of the default color (black or white), and similarly depth buffer will be cleared to the default value of 1.0 (the farthest depth value for the pixel). Consider objects are rendered in the order of cylinder, sphere, and cone as given in the diagram. The table given below sums up how the pixel P and its depth value d will be updated after every object is drawn; a similar process happens for all the remaining pixels.

Operation	***P***	***d***	**Description**
Clear Operation	Black	1.0	Pixel and corresponding depth entry initialized.
Draw Cylinder	P_3	d_3	Since $d_3 <= d = 1.0$ the depth test passes and we update the buffers by setting $P = P_3$ and $d = d_3$
Draw Sphere	P_1	d_1	Since $d_1 <= d = d_3$ the depth test passes and we update the buffers by setting $P = P_1$ and $d = d_1$
Draw Cone	P_1	d_1	Since $d_2 > d = d_1$ the depth test fails and we do not update the buffers.

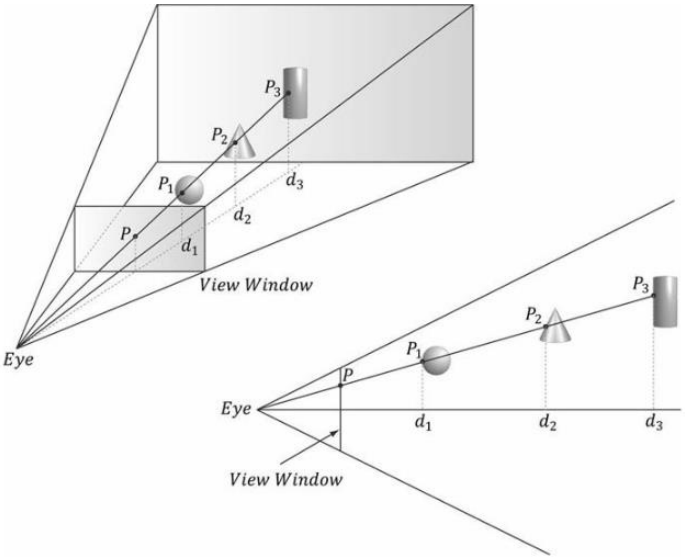


Figure 5.3. The view window corresponds to the 2D image (back buffer) we generate of the 3D scene. We see that three different pixels can be projected to the pixel P. Intuition tells us that P1 should be written to P because it is closer to the viewer and blocks the other two pixels. The depth buffer algorithm provides a mechanical procedure for determining this on a computer. Note that we show the depth values relative to the 3D scene being viewed, but they are actually normalized to the range [0.0, 1.0] when stored in the depth buffer.

We only update the pixel and its related depth value in the depth buffer as we find a pixel with a smaller depth value.

The closest pixel from the viewer will be the only one which will be rendered. You can check by updating the values in the table by shuffling the order of drawing of the objects.

The depth buffering computes a depth value for every pixel in the frame and performs a depth test. This depth test is used to compares depths of pixels which are competing to be written to a particular pixel position on the back buffer. The pixel which has the depth value closest to the viewer will be drawn on that position, and that pixel that gets written to the back buffer. Is simply means pixel closest to the viewer will hide or obscure the pixels behind it.

Depth buffer is a texture and there are some specific formats used for drawing the same. The formats are as follows:

DXGI_FORMAT_D32_FLOAT_S8X24_UINT	It specifies a 32-bit floating-point depth buffer, with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range and 24-bits not used for padding.
DXGI_FORMAT_D32_FLOAT	It specifies a 32-bit floating-point depth buffer.

DXGI_FORMAT_D24_UNORM_S8_UINT	It specifies an unsigned 24-bit depth buffer mapped to the [0, 1] range with 8-bits (unsigned integer) reserved for the stencil buffer mapped to the [0, 255] range.
DXGI_FORMAT_D16_UNORM	It specifies an unsigned 16-bit depth buffer mapped to the [0, 1] range.

5.6 TEXTURE RESOURCE VIEWS

A texture is bound to different stages in the rendering pipeline; a simple example is to use a texture as the render target and as the shader resource (i.e., here the texture will be sampled in a shader). Texture resource can be created for these two things would be given with following bind flags:

D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE

which indicates the two pipeline stages that the texture will be bound to. Here the resources are not directly bound to any pipeline stage; instead their associated resource views are bound to different pipeline stages.

We will use a texture in both the cases, Direct3D requires us to create a resource view of that texture at the time of initialization. This is done for efficiency and mentioned in the SDK documentation as: "This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time."

Hence to use a texture as a render target and shader resource, we need to create two views: a render target view (**ID3D11RenderTargetView**) and a shader resource view (**ID3D11ShaderResourceView**).

Two things resource views always do: 1) they tell Direct3D how the resource will be used means, what stage of the pipeline you will bind it to, and 2) if the resource format was specified as a *typeless* at the time of creation, then we must now assign a type while creating the view. With typeless formats, we can view elements of a texture as the floating-point values in one pipeline stage and as the integers in other. To create a specific view to a resource, we need to create resources with that specific bind flag. For example, if we won't create the resource with the **D3D11_BIND_DEPTH_STENCIL** bind flag (this indicates the texture will be bound to the pipeline as a depth/stencil buffer), after this we cannot create an **ID3D11DepthStencilView** to that resource.

If you try, you should get a Direct3D debug error like the following:

D3D11: ERROR: ID3D11Device::CreateDepthStencilView: A DepthStencilView cannot be created of a Resource that did not specify D3D11_BIND_DEPTH_STENCIL.

Multisampling Theory:

As the pixels on our display monitor are not infinitely small, the arbitrary line can not be displayed perfectly on such monitors. Figure 5.4 given below illustrates a "stair-step" or aliasing effect, which occurs when approximating a line by a matrix of pixels. Similar kind of aliasing effects can also occur with edges of triangles as well.

Figure 5.4. On the top we observe aliasing (the stair-step effect when trying to represent a line by a matrix of pixels). On the bottom, we see an antialiased line, which 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

One way to overcome this effect is to shrink the pixel size by increasing the monitor resolution which me resolve the issue and the stair-step effect may not be noticed by users. In the cases where this solution will not work, we must use antialiasing techniques.

One such technique, called as *supersampling*, makes the back buffer and depth buffer 4 times bigger than the screen resolution. Then the 3D scene is will be rendered to the back buffer at this large resolution. At the time to present the back buffer to the screen, the back buffer is resolved or downsampled such that the 4 pixel block colors will be averaged together to get an averaged value of pixel color. Here, supersampling actually works by increasing the screen resolution in software.

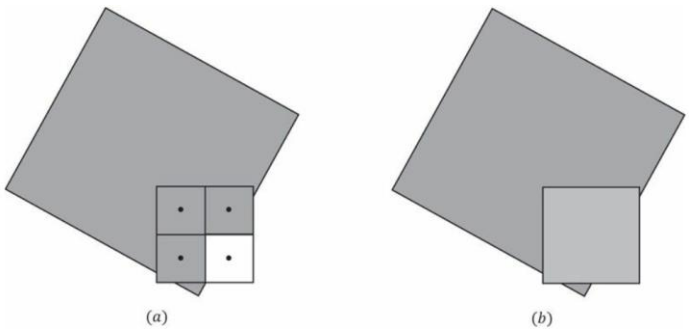


Figure 5.5. We consider one pixel that crosses the edge of a polygon. (a) The green color evaluated at the pixel center is stored in the three visible subpixels that are covered by the polygon. The subpixel in the 4th quadrant is not covered by the polygon and so does not get updated with the green color–it just keeps its previous color computed from previously drawn geometry or the Clear operation. (b) To compute the resolved pixel color, we average the four subpixels (three green pixels and one white pixel) to get a light green

along the edge of the polygon. This results in a smoother looking image by diluting the stair-step effect along the edge of the polygon.

Supersampling technique is expensive, as it increases the amount of pixel processing and memory four times as it increases resolution. Direct3D supports a mild antialiasing technique called *multisampling*, which actually shares some of the computational information with subpixels making it less expensive in terms of processing than supersampling.

Assuming we are using 4 times multisampling (4 subpixels/pixel), multisampling also uses a back buffer and depth buffer 4 times bigger than the given screen resolution. Rather than computing the image color for each given subpixel, multissampling computes it only one time per pixel, at the pixel center, and then shares that color information with its all subpixels based on their visibility (the depth/stencil test is evaluated per subpixel) and coverage (does the subpixel center lie inside or outside the polygon?). Figure 5.5 shows an example.

Difference between Multisampling and Supersampling is given below:

Supersampling	**Multisampling**
Here the image color is computed per subpixel, hence having a different color	Here image color is computed only once per pixel and that color is replicated into all visible subpixels
It is technically more accurate and handles texture and shader aliasing	Multisampling is not accurate and handles texture and shader aliasing
It is expensive	It is not expensive

5.8 MULTISAMPLING IN DIRECT3D:

Now we will be required to fill out a **DXGI_SAMPLE_DESC** structure. It has two members and is defined like below:

typedef struct DXGI_SAMPLE_DESC {

UINT Count;
UINT Quality;

} DXGI_SAMPLE_DESC, *LPDXGI_SAMPLE_DESC;

Here **count** is used to specify the number of samples to be taken per pixel, and **Quality** member is used for specifying the quality level desired. *Quality levels may vary based on the hardware manufacturers.* Sample counts and quality level which are higher may cost expensive in rendering, so we need to choose between quality and speed. Quality ranges has many levels that depend on the texture format and the number of samples to be taken per pixel. We can use the following method to query the quality levels for sample count and texture format:

HRESULT ID3D11Device::CheckMultisampleQualityLevels(

DXGI_FORMAT Format, UINT SampleCount, UINT *pNumQualityLevels);

It returns zero if the format and sample count combination is not supported by the given device. Else, **pNumQualityLevels parameter** will be used to return the number of quality levels for the given combination. The valid quality level range is from zero to **pNumQualityLevels −1**.

Maximum number of samples that can be taken per pixel is defined by the preprocessor directive in C/C++ as:

#define D3D11_MAX_MULTISAMPLE_SAMPLE_COUNT (32)

A sample count of 4 or 8 is common to keep the performance and memory cost of multisampling reasonable. If we are not using multisampling, we can set the sample count to one and the quality level to zero. All the devices which are Direct3D 11 capable support 4 times multisampling for all kinds of render target formats.

5.9 FEATURE LEVELS

Feature levels concept is introduced in Direct3D 11 which is represented in code by the using the **D3D_FEATURE_LEVEL** enumerated type, which corresponds to various Direct3D versions ranging from version 9 to 11:

```
typedef enum D3D_FEATURE_LEVEL
{
D3D_FEATURE_LEVEL_9_1 = 0x9100,
D3D_FEATURE_LEVEL_9_2 = 0x9200,
D3D_FEATURE_LEVEL_9_3 = 0x9300,
D3D_FEATURE_LEVEL_10_0 = 0xa000,
D3D_FEATURE_LEVEL_10_1 = 0xa100,
D3D_FEATURE_LEVEL_11_0 = 0xb000,
} D3D_FEATURE_LEVEL;
```

A strict set of functionalities are defined in feature levels which are specified in the SDK documentation, for the specific capabilities of each feature level. In case of a user device not supporting a given feature level, the application falls back to the older feature level.

For example, as supported by devices of large audience, some application might support Direct3D 11, 10.1, 10, and 9.3 level hardware. Usually any application will check the support for feature levels from newest to oldest: means, the application may first check if Direct3D 11 is supported, second Direct3D 10.1, then Direct3D 10, and finally Direct3D 9.3. The following feature level array may be used for supporting the order of testing:

```
D3D_FEATURE_LEVEL featureLevels[4] =
{
D3D_FEATURE_LEVEL_11_0, // First check D3D 11 support
D3D_FEATURE_LEVEL_10_1, // Second check D3D 10.1 support
D3D_FEATURE_LEVEL_10_0, // Next, check D3D 10 support
D3D_FEATURE_LEVEL_9_3 // Finally, check D3D 9.3 support
};
```

Direct3D initialization function will take this array as an input, and the output will be the first supported feature level in the array as calculated by the function. For example, consider Direct3D reported back that the first feature level in the array that was supported as **D3D_FEATURE_LEVEL_10_0**, then that application could disable Direct3D 11 and Direct3D 10.1 features and use the Direct3D 10 features for rendering path. In this text, we are considering the support of **D3D_FEATURE_LEVEL_11_0**, as our focus is on Direct3D 11. We need to keep in mind that, real-world applications may not worry about supporting the older hardware to support the wide array of audience.

5.10 QUESTIONS:

1. Explain DirectX.
2. What are textures and data resource formats?
3. Explain swap chains and page flipping.
4. What is depth buffering?
5. What is texture resource view?
6. What is Multisampling?
7. What are feature levels?

❄❄❄❄❄❄❄

6

DIRECT3D 11 RENDERING PIPELINE

Unit Structure :

INTRODUCTION

Rendering pipeline is the core and main concept to be understood first along with its stages. In the geometric and graphical description of a 3D scene with a virtual camera which is positioned and oriented, the pipeline refers entire sequence of steps necessary to create the 2 dimensional image as what virtual camera sees (Figure 6.1).

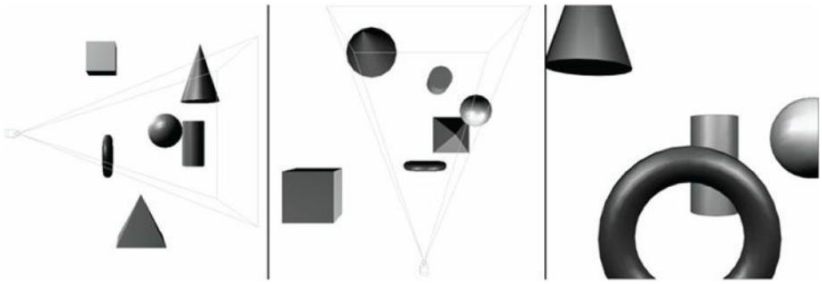


Figure 6.1. The left image shows a side view of some objects set up in the 3D world with a camera positioned and aimed; the middle image shows the same scene, but from a top-down view. The "pyramid" volume specifies the volume of space that the viewer can see; objects (and parts of objects) outside this volume are not seen. The image on the right shows the 2D image created based on what the camera "sees."

6.0 OBJECTIVES:

1. To understand the rendering pipeline—the process of taking a geometric description of a 3D scene and generating a 2D image from it.
2. To learn how to specify the part of a texture that gets mapped to a triangle.
3. To find out how to create and enable textures.
4. To learn how textures can be filtered to create a smoother image.
5. To gain a basic understanding of the interaction between lights and materials.
6. To understand the differences between local illumination and global illumination.
7. To find out how we can mathematically describe the direction a point on a surface is "facing" so that we can determine the angle at which incoming light strikes the surface.
8. To understand how blending works and how to use it with Direct3D.
9. To learn about the different blend modes that Direct3D supports.

6.1 OVERVIEW

Rendering pipeline is generally the entire sequence of steps, which is necessary to generate a 2D image based on what virtual camera is able to capture which is positioned and oriented in a 3D scene.

In Figure 6.2, we can see the diagram showing connections of stages of the rendering pipeline, it also includes GPU memory resources off to the side. Arrows are used to indicate the directions of data and information flow. Like an arrow from the resource memory pool to a stage means that stage

can access the resources as input; for example, in the pixel shader stage, it may need to read data from a texture resource stored in the memory in order to do its work. In the case of an arrow going from a stage to memory means the stage writes something to GPU resources; for example, in the output merger stage, it writes data to textures like the back buffer and depth/stencil buffer.

The arrow for the output merger stage is bidirectional (means it reads from and writes to GPU resources). Most stages in the pipeine do not write to GPU resources. Instead, their output is just given in as an input to the next stage in the pipeline; for example, see the Vertex Shader stage inputs data from the Input Assembler stage, performs the work, and then outputs its results to the Geometry Shader stage in pipeline. The next sections will explain each stage in detail:

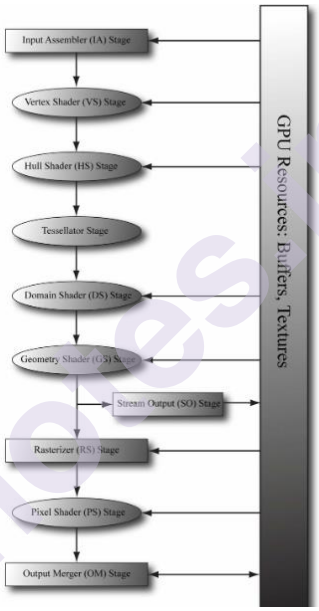


Figure 6.2. The stages of the rendering pipeline.

6.2 INPUT ASSEMBLER STAGE (IA)

The input assembler (IA) stage takes geometric data (i.e vertices and indices) from memory resources and uses it to calculate and assemble geometric primitives (e.g., triangles, lines, dots). Triangles and lines are basic building blocks for animation/geometry in graphics.

6.2.1 Vertices

The vertices of a triangle are where two edges meet according to mathematics; the vertices of a line are the endpoints connecting each other; for a single point, the point itself is considered as the vertex.

In Figure 6.3 we can see vertices in pictorial form. It shows that a vertex is just some special point in a geometric primitive. In Direct3D, vertices are considered as much more general.

A vertex in Direct3D may also consist of additional data apart from the spatial (imagery) location, which allows the programmer to perform some of the more sophisticated rendering effects and transitions. In Direct3D we have the flexibility to create and define our own vertex formats (i.e., it allows us to define the components of a vertex).

6.2.2 Primitive Topology

All vertices in IA stage are bound to the rendering pipeline in one specially created Direct3D data structure called as a vertex buffer. It is used for storing the list of vertices in a contiguous memory. But it doesn't say anything about how to arrange these vertices to form geometric primitives. It means, we can't say that every three vertices will form a triangle and every set if two vertices will form a line. To serve the purpose of telling Direct3D about how to draw the geometry by using vertices, we use *primitive topology.*

The code example and syntax for the same are given below:

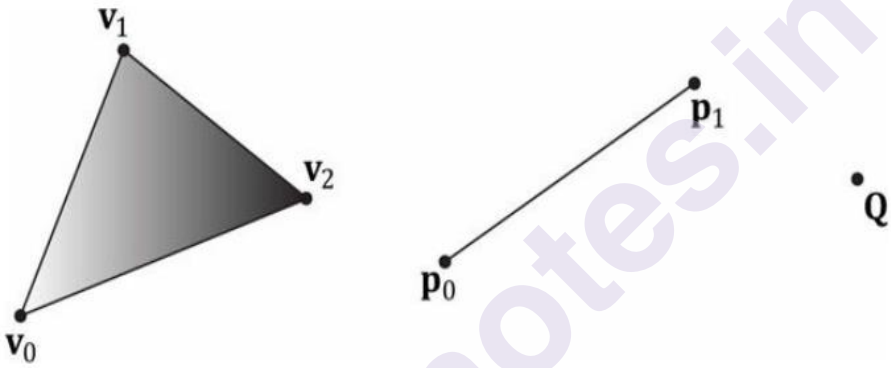


Figure 6.3. A triangle defined by the three vertices v0, v1, v2; a line defined by the two vertices p0, p1; a point defined by the vertex Q.

```
void ID3D11DeviceContext::IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY Topology);
typedef enum D3D11_PRIMITIVE_TOPOLOGY
{

D3D11_PRIMITIVE_TOPOLOGY_UNDEFINED = 0,
D3D11_PRIMITIVE_TOPOLOGY_POINTLIST = 1,
D3D11_PRIMITIVE_TOPOLOGY_LINELIST = 2,
D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP = 3,
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST = 4,
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP = 5,
D3D11_PRIMITIVE_TOPOLOGY_LINELIST_ADJ = 10,
D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ = 11,
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ = 12,
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ = 13,
D3D11_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCH
LIST = 33,
```

D3D11_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCH LIST = 34,
.
.
.
D3D11_PRIMITIVE_TOPOLOGY_32_CONTROL_POINT_PATCH LIST = 64,

} D3D11_PRIMITIVE_TOPOLOGY;

All subsequent drawing calls will use the currently set primitive topology until the topology is changed. The following code illustrates:

```
md3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_LINELIST);
/* ...draw objects using line list... */

md3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
/* ...draw objects using triangle list... */

md3dImmediateContext->IASetPrimitiveTopology(
D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
/* ...draw objects using triangle strip... */
```

In next subsections, we will understand different types of primitive topologies:

6.2.2.1 Point List

A point list is given by the code line **D3D11_PRIMITIVE_TOPOLOGY_POINTLIST**. Every vertex in the draw call is drawn as an individual point while using point list, it is shown in Figure 6.4a.

6.2.2.2 Line Strip

A line strip is given by the code line **D3D11_PRIMITIVE_TOPOLOGY_LINESTRIP**. The vertices in the draw call are connected to form lines while using line strip (see Figure 6.4 b); here n + 1 vertices induces n lines.

6.2.2.3 Line List

A line list is given by code line **D3D11_PRIMITIVE_TOPOLOGY_LINELIST**. Every two vertices in the draw call forms an individual line while using line list (see Figure 6.4c); so here 2n vertices induces n lines. The main difference in line list and line strip is that, in line list the lines may be disconnected whereas in line strip makes them automatically connected; by this connectivity, fewer vertices can be used because every interior vertex is shared by two lines.

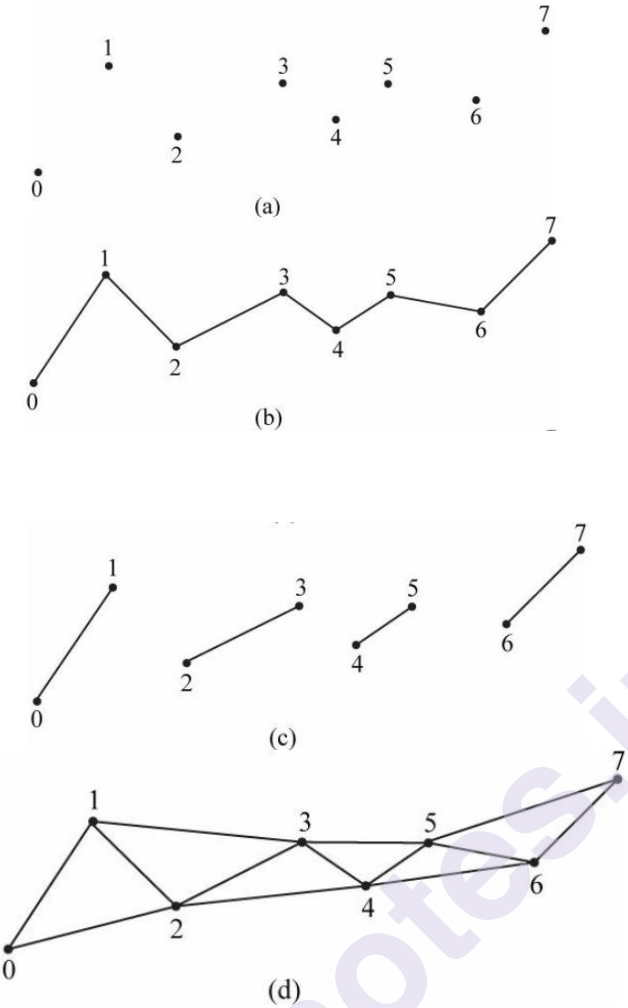


Figure 6.4. (a) A point list; (b) a line strip; (c) a line list; (d) a triangle strip.

6.2.2.4 Triangle Strip

A triangle strip is given by code line **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP**. It is assumed the triangles are connected while using Triangle Strip as shown in Figure 6.4d to form a strip. Connectivity is assumed, and we see that vertices are shared between two adjacent triangles, and n vertices induce n – 2 triangles.

6.2.2.5 Triangle List

A triangle list is given by code line **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST**. Every three vertices in the draw call forms an individual triangle while using triangle list (see Figure 6.5a); so 3n vertices induces n triangles. The main difference between triangle list and triangle strip is that, in triangle list the triangles may or may not be connected but in case of triangle strip, as we have seen, triangles are connected automatically.

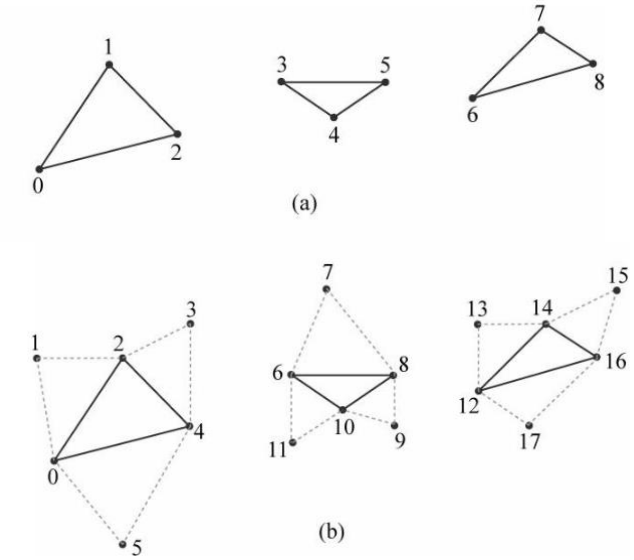


Figure 6.5. (a) A triangle list. (b) A triangle list with adjacency—observe that each triangle requires 6 vertices to describe it and its adjacent triangles. Thus 6n vertices induces n triangles with adjacency info.

6.2.2.6 Primitives with Adjacency

We can have a triangle list with adjacency where, for every triangle, we include data of the three adjacent triangles (one for each side); in Figure 6.5b you can observe how these triangles are defined.

This technique is used in the geometry shader, where some geometry shading algorithms need an access to the adjacent triangles. For this purpose the geometry shader submits the adjacent triangles to the pipeline in the vertex/index buffer and the triangle itself, and the **D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ** topology has to be specified because, the pipeline knows how to construct the triangle and its adjacent triangles from the vertex buffer. We have to note that the vertices of adjacent primitives are only used as input into the geometry shader—they are not drawn. Without the geometry shader, the adjacent primitives are still not drawn. We can also have a line list with adjacency, line strip with adjacency, and triangle with strip adjacency primitives; all the details are given in SDK documentation.

6.2.2.7 Control Point Patch List

The code **D3D11_PRIMITIVE_TOPOLOGY_N_CONTROL_POINT_PATCH LIST** topology type is used for indicating that the vertex data should be interpreted as a patch lists with N number of control points. These will be optionally used in tessellation stage of the rendering pipeline.

6.2.3 Indices

As we know, triangles are the basic building blocks for the solid 3D objects. The code given below shows the vertex arrays used to construct a quad and octagon using triangle lists (i.e., every three vertices form a triangle).

```
Vertex quad[6] = {

v0,        v1,        v2,        //        Triangle        0
v0, v2, v3, // Triangle 1

};
Vertex octagon[24] = {

v0,        v1,        v2,        //        Triangle        0
v0,        v2,        v3,        //        Triangle        1
v0,        v3,        v4,        //        Triangle        2
v0,        v4,        v5,        //        Triangle        3
v0,        v5,        v6,        //        Triangle        4
v0,        v6,        v7,        //        Triangle        5
v0,        v7,        v8,        //        Triangle        6
v0, v8, v1 // Triangle 7

};
```

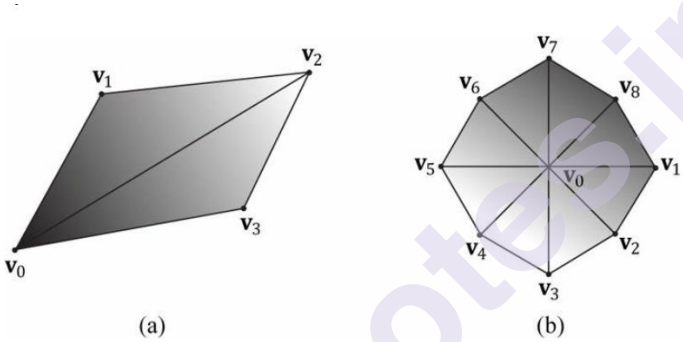


Figure 6.6. (a) A quad built from two triangles. (b) An octagon built from eight triangles.

As given in figure 6.6, the triangles which are forming a 3D object share lot of the same vertices. Specifically speaking, each triangle of the quad in Figure 5.15a shares the vertices v0 and v2. Here, the duplication is worse in the octagon example (Figure 6.6b), because each triangle in the diagram duplicates the center vertex v0, also each vertex on the perimeter of the octagon is shared by two adjacent triangles.

Generally speaking, the number of duplicate vertices increases as the detail and complexity of the model increases. We need to take into consideration two reasons why we should not duplicate the vertices:

1. Increased memory requirements. (Why store the same vertex data more than once?)

2. Increased processing by the graphics hardware. (Why process the same vertex data more than once?)

We can use triangle strips to solve this problem, given that, the geometry can be organized in a strip-like fashion (which may not be the case always). As we know, triangle lists are more flexible because the triangles need not be connected, and so it is worth creating a method to remove duplicate

vertices for triangle lists. The better solution is to use indices. It works as follows:

We first create a vertex list and a matching index list. Then the vertex list consists of all the unique vertices and the index list contains only values that index into the vertex list to define how the vertices are to be put together to form triangles. The vertex list of the quad can be constructed as follows:

Vertex v[4] = {v0, v1, v2, v3};

Then the index list needs to define how the vertices in the vertex list are to be put together to form the two triangles.

UINT indexList[6] = {0, 1, 2, // Triangle 0
0, 2, 3}; // Triangle 1

In the index list, every three elements define a triangle. So the previous index list says, "form triangle 0 by using the vertices v[0], v[1], and v[2], and form triangle 1 by using the vertices v[0], v[2], and v[3]."Similarly, the vertex list for the circle would be constructed as follows:

Vertex v [9] = {v0, v1, v2, v3, v4, v5, v6, v7, v8};

and the index list would be:

UINT indexList[24] = {

0, 1, 2, // Triangle 0
0, 2, 3, // Triangle 1
0, 3, 4, // Triangle 2
0, 4, 5, // Triangle 3
0, 5, 6, // Triangle 4
0, 6, 7, // Triangle 5
0, 7, 8, // Triangle 6
0, 8, 1 // Triangle 7

};

Once we process unique vertices, index list is used by the graphics card to put the vertices together to form the triangles. Here we have successfully moved the duplication to index list, this will not cause a problem because:

1. Indices are simply integers and do not take up as much memory as a full vertex structure (and vertex structures can get big as we add more components to them).

2. With good vertex cache ordering, the graphics hardware won't have to process duplicate vertices (too often).

6.3 VERTEX SHADER STAGE (VS)

After the IA stage where primitives have been assembled, the created vertices are fed into the next stage i.e vertex shader (VS).

You can think of a vertex shader stage as a function which takes a vertex as input parameter and also outputs a vertex. Each vertex drawn will be pumped through the vertex shader. We can understand the working of this function by using following code:

for(UINT i = 0; i < numVertices; ++i)

outputVertex[i] = VertexShader (inputVertex[i]);

Remember vertex shader function is implemented by us, but GPU will execute for every vertex in the diagram, so it is very fast. Vertex shader is used for creating various special effects line transformations, lighting, and displacement mapping. Here along with the access to the input vertex data, we also can access textures and other data which is stored in GPU memory as transformation matrices and scene lights. Now we will understand the kinds of transformations that needed to be done using vertex shader stage in following subsections:

6.3.1 Local Space and World Space

Most of the times when you are working on a scene, consider for creating a movie, you create small properties and once they are perfectly built, you can put them into the main scene.

3D artists or programmers do something similar when constructing 3D objects. Instead of working in the global scene coordinate system (world space) they specify the things into the local scene coordinate system (local space); This local coordinate system is related with the coordinate system aligned to the geometry of an object instead of the whole scene's geometry, hence it is very easy to work on that first.

After the vertices in the local scene are created, we can go put the object into the global system (world space). To achieve this, we need to understand how the local space and world space are related; this is done by specifying where we want the origin and axes of the local space coordinate system relative to the global scene coordinate system. Then we perform the change of coordinate transformation as it is given in Figure 6.7. This entire process of changing coordinates related to local system into the global system is known as the world transform, and the matrix used in this process is called as the world matrix. Every object in the given scene has its own different world matrix. After the transformation of every object from its local space to the world space, all the coordinates of every object are related with the world space. We can define an object directly into the world space by using identity world matrix as the coordinate system. Using local coordinate system for each object is advantageous in following ways:

1. It is easier. We know that, usually in local space the object will be created as centered at the origin and symmetrical with respect to one of it's major axes. For example, the vertices of a cube are much easier to specify if we choose a local coordinate system with origin, which is centered at the cube and with the axes orthogonal to the cube faces; see Figure 6.8.

2. We can use the same objet in multiple scenes according to our needs, hence we can not hardcode the object's coordinates to a particular scene. It is better to store its coordinates relative to a local coordinate system for that object and use the coordinate matrix, and transfer the object to the required scene as per the need.

3. In some cases, we draw the same object multiple times in the same scene, but in different positions, orientations, and scales (e.g., if we are creating an animated forest then the tree object may be drawn several times with different shapes, sizes and positions). We can store a single copy of the geometry (i.e., vertex and index lists) relative to its local space. For several times we can draw the object afterwards, but the world matrix will be different each time to specify the position, orientation, and scale of the object's instance in the world space. This is called *instancing*.

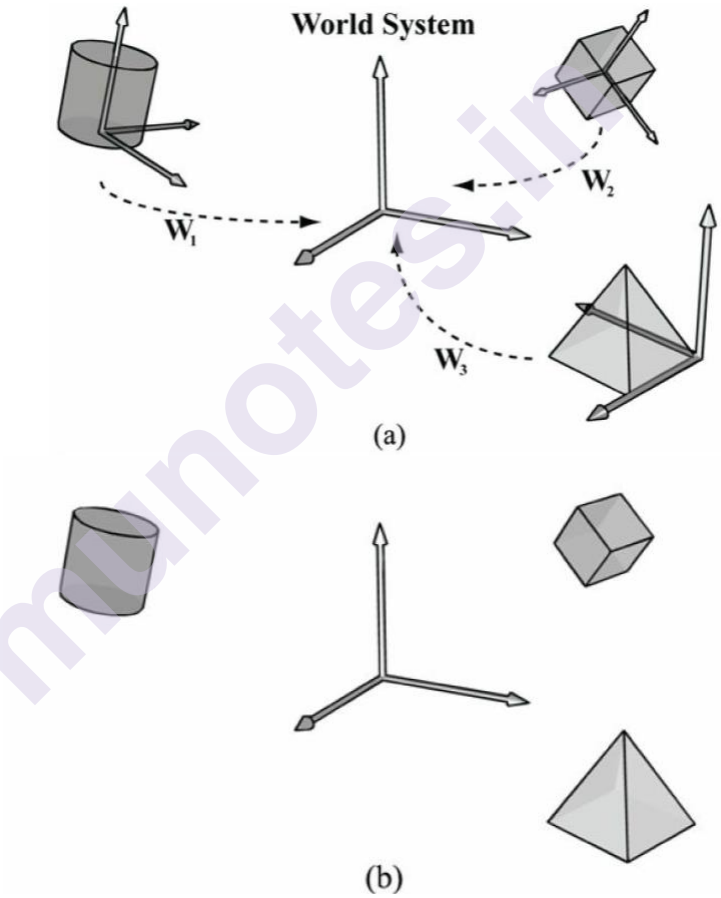


Figure 6.7. (a) The vertices of each object are defined with coordinates relative to their own local coordinate system. In addition, we define the position and orientation of each local coordinate system relative to the world space coordinate system based on where we want the object in the scene. Then we execute a change of coordinate transformation to make all coordinates relative to the world space system. (b) After the world transform, the objects' vertices have coordinates all relative to the same world system.

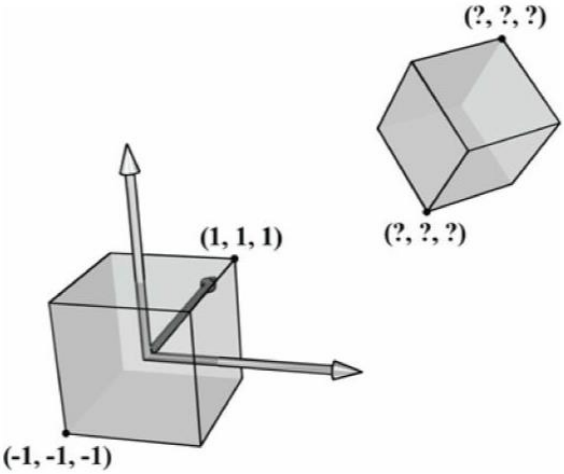


Figure 6.8. The vertices of a cube are easily specified when the cube is centered at the origin and axis-aligned with the coordinate system. It is not so easy to specify the coordinates when the cube is at an arbitrary position and orientation with respect to the coordinate system. Therefore, when we construct the geometry of an object, we usually always choose a convenient coordinate system near the object and aligned with the object, from which to build the object around.

The Matrix Representation: Qw = (Qx ,Qy ,Qz ,1), uw = (ux , uy , uz ,0), vw = (vx ,vy ,vz ,0), and ww = (wx , wy , wz ,0) describe the origin and axes of frame A with homogeneous coordinates relative to frame B. This 4 × 4 matrix is called as a change of coordinate matrix or change of frame matrix, and it converts (or maps) frame A coordinates into frame B coordinates. The world matrix for an object is given as a description of its local space with coordinates relative to the world space, and placing these coordinates as the rows of a matrix. If Qw = (Qx, Qy, Qz, 1), uw = (ux, uy, uz, 0), vw = (vx, vy, vz, 0), and ww = (wx, wy, wz, 0) describe, respectively, the origin, x-, y-, and z-axes of a local space with homogeneous coordinates relative to world space, then we know that the change of coordinate matrix from local space to world space is

$$\mathbf{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

We need to figure out the local space origin coordinates and axes which are relative to world space. It is not very easy.

One common approach can be taken that is to define W as a sequence of transformations, say W = SRT, it is the product of a scaling matrix S to scale the object into the world, followed by a rotation matrix R to define the orientation of the local space relative to the world space, followed by a translation matrix T to define the origin of the local space relative to the

world space. This sequence of transformations may be interpreted as a change of coordinate transformation, and that the row vectors of W = SRT store the homogeneous coordinates of the x-axis, y-axis, z-axis, and origin of the local space relative to the world space.

6.3.2 View Space

We place the virtual camera into the space to form the 2D image of the scene. This camera specifies what volume and size of the world the viewer can see and thus what volume of the world we need to generate using the 2D image.

As shown in Figure 6.9 attach the local coordinate system to the virtual camera; that is, the camera will be located the origin looking down the positive z-axis, the x-axis aims to the right of the camera, and the y-axis aims above the camera.

It is beneficial to describe our scene vertices relative to the camera coordinate system in rendering pipeline rather than describing them relative to the world space. This change of coordinate transformation from world space to view space (camera space) is called as the view transform, and the corresponding matrix is called the *view matrix*.

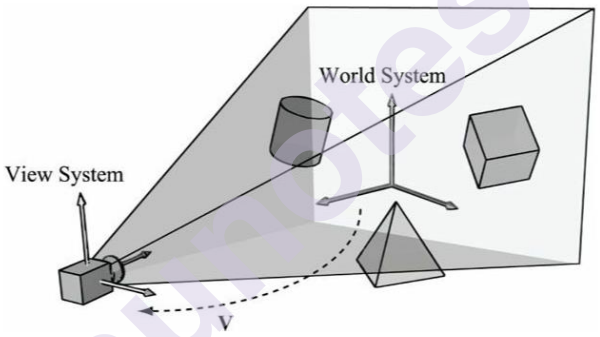


Figure 6.9. Convert the coordinates of vertices relative to the world space to make them relative to the camera space.

If $Q_w = (Q_x, Q_y, Q_z, 1)$, $uw = (u_x, u_y, u_z, 0)$, $vw = (v_x, v_y, v_z, 0)$, and $ww = (w_x, w_y, w_z, 0)$ describe, respectively, the origin, x-, y-, and z-axes of view space with homogeneous coordinates relative to world space, then the change of coordinate matrix from view space to world space is given as:

$$W = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_y & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

This is not the required transformation. We need the transformation from world space to view space, the reverse transformation. We can achieve this by using the inverse of matrix, W^{-1} transforms from world space to view space. The world and view coordinate systems differ in the position and orientation properties only, so it makes intuitive The world coordinate system and view coordinate system generally differ by position and

orientation only, so it makes intuitive sense that W = RT (i.e., the world matrix can be decomposed into a rotation followed by a translation). The inverse form can be computer easily as:

$V=W^{-1} = (RT)^{-1} = T^{-1} R^{-1} = T^{-1} R^{T}$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_y & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}$$

$$= \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_y & 0 \\ w_x & w_y & w_z & 0 \\ -Q.u & -Q.v & -Q.w & 1 \end{bmatrix}$$

So the view matrix has the form:

$$V = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_y & 0 \\ w_x & w_y & w_z & 0 \\ -Q.u & -Q.v & -Q.w & 1 \end{bmatrix}$$

Another way to construct the vectors needed to build the view matrix.

Now, let Q be the position of the camera and let T be the target point the camera is aimed at. Then, let j be the unit vector that describes the "up" direction of the world space. (we generally consider the world xz-plane as our world "ground plane" and the world y-axis describes the "up" direction; therefore, j = (0,1,0) is just a unit vector parallel to the world y-axis.)

By the reference to Figure 6.10, the direction the camera is looking is given by:

$$w = \frac{\mathrm{T} - \mathrm{Q}}{|\mathrm{T} - \mathrm{Q}|}$$

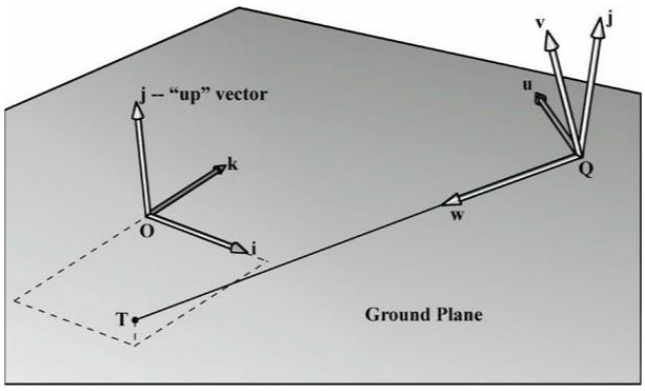


Figure 6.10. Constructing the camera coordinate system given the camera position, a target point, and a world "up" vector.

This vector describes the local z-axis of the camera. A unit vector that aims to the "right" of w is given by:

$$u = \frac{j \times w}{| j \times w |}$$

The local x-axis of camera in defined by this vector. The vector that defines y-axis of the camera is given by:

v = w × u

Here w and u are the orthogonal unit vectors, w × u is necessarily a unit vector (by convention), and so it does not need to be normalized. Given the position of the camera, the target point, and the world "up" direction, we can derive the local coordinate system of the camera, which can be used to form the view matrix. The following function provided by the XNA mathematics library is used for computing the view matrix based on the just described process:

XMMATRIX XMMatrixLookAtLH(// Outputs resulting view matrix V
FXMVECTOR EyePosition, // Input camera position Q
FXMVECTOR FocusPosition, // Input target point T
FXMVECTOR UpDirection); // Input world up vector j

Usually the world's y-axis corresponds to the "up" direction, so the "up" vector is almost always j = (0,1,0). As an example, suppose we want to position the camera at the point (5, 3, −10) relative to the world space, and have the camera look at the origin of the world (0, 0, 0). We can build the view matrix by writing:

```
XMVECTOR pos = XMVectorSet(5, 3, -10, 1.0f);
XMVECTOR target = XMVectorZero();
XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
XMMATRIX V = XMMatrixLookAtLH(pos, target, up);
```

6.3.3 Projection and Homogeneous Clip Space

Yet we have seen the position and orientation of the camera, one other important component of camera we need to take into consideration, which is the volume of space the camera sees. A frustum is used to describe this volume as given in Figure 6.11.

Then we project this frustum from the 3D scene on to the 2D projection window. This projection must be done in a certain way, that is the parallel lines must converge to a vanishing point, similarly as the 3D depth of an object increases, the size of its projection must diminish; a perspective projection does this, and is illustrated in Figure 6.12.

This line from vertex to the eye point is known as vertex's line of projection. After this we define the perspective projection transformation as the transformation that is used to transform a 3D vertex v to the point v′ where its line of projection intersects the 2D projection plane; we say that v′ is the

projection of v. The projection of a 3D object is the projection of all the vertices those which make up that object.

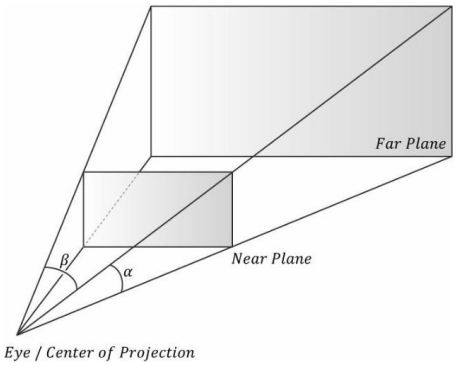


Figure 6.11. A frustum defines the volume of space that the camera "sees."

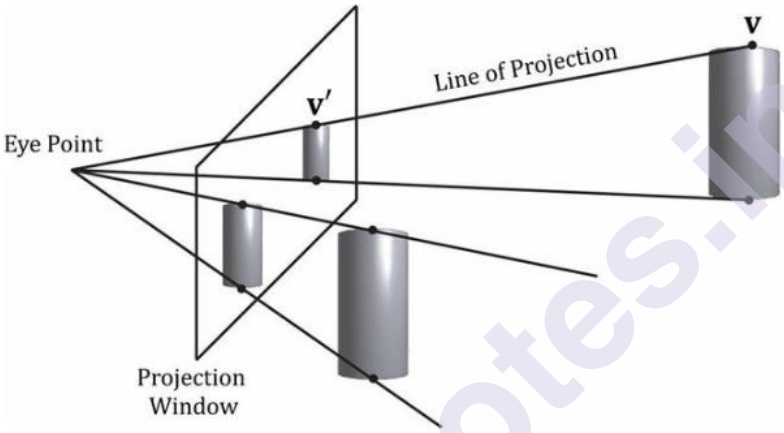


Figure 6.12. Both cylinders in 3D space are the same size but are placed at different depths. The projection of the cylinder closer to the eye is bigger than the projection of the farther cylinder. Geometry inside the frustum is projected onto a projection window; geometry outside the frustum gets projected onto the projection plane but will lie outside the projection window.

6.3.3.1 Defining a Frustum

Frustum, as we know, is defined in the view space, where the center of projection at the origin and looking down the positive z-axis, by the following four quantities: A near plane 'n', a far plane 'f', a vertical field of view angle 'a', and an aspect ratio 'r'.

Note that in view space, the near plane and far plane are parallel to the xy-plane; hence we can simply specify their distance from the origin along the z-axis. We can use the equation r = w/h to define the aspect ratio where w is the width of the projection window and h is the height of the projection window (units in view space). In view space, the projection window that we are considering is a 2D image. This image here will eventually be mapped to the back buffer; hence, we like the ratio of the projection window dimensions to be the same as the ratio of the back buffer dimensions. This ratio of the back buffer dimensions is called as an aspect ratio (it is a ratio

so it has no units). For example, if the back buffer dimensions are 800 × 600, then we specify. $r = \frac{800}{600} \approx 1.333$. In the case where the aspect ratio of back buffer and projection window is not same, then a nonuniform scaling becomes necessary to map the projection window to the back buffer, which would cause a sort of distortion (e.g., a circle on the projection window might get stretched into an ellipse when mapped to the back buffer).

The horizontal field of view angle is labeled as β, and it is determined by the vertical field of view angle as α and aspect ratio as r. See Figure 6.13 to understand how r is used to find the value of β. Here the actual dimensions of the projection window are not important, what is important is the aspect ratio needs to be maintained. Hence, we will choose the convenient height of 2, and thus the width must be:

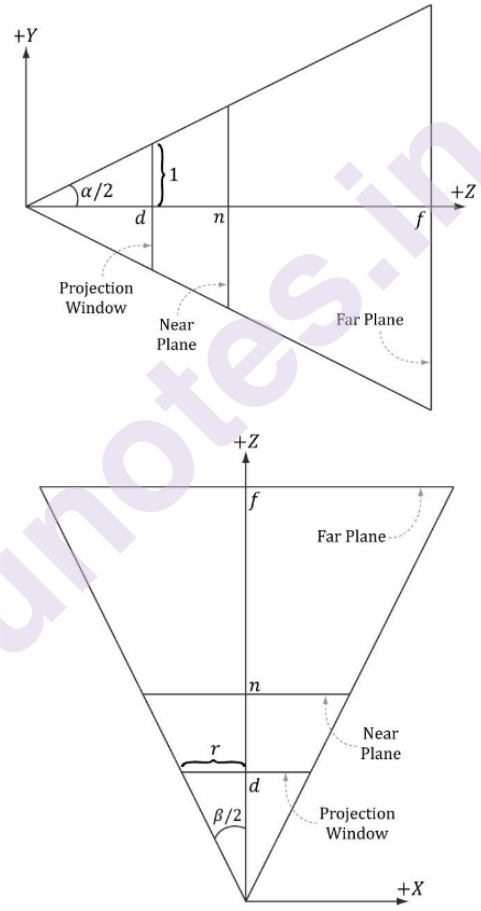


Figure 6.13. Deriving the horizontal field of view angle β given the vertical field of view angle a and the aspect ratio r.

$$r = \frac{w}{h} = \frac{w}{2} \Rightarrow w = 2r$$

To have the specified vertical field of view a, the projection window must be placed a distance d from the origin:

$$tan\left(\frac{\alpha}{2}\right) = \frac{1}{d} \Rightarrow d = cot\left(\frac{\alpha}{2}\right)$$

Here we have fixed the distance d of the projection window along the z-axis to have a vertical field of view a when the height of the projection window is 2. Now we can solve for β. By Figure 6.13 given the xz-plane, we now see that:

$$tan\left(\frac{\beta}{2}\right) = \frac{r}{d} = \frac{r}{cot\left(\frac{\alpha}{2}\right)}$$

$$= r.tan\left(\frac{\alpha}{2}\right)$$

So given the vertical field of view angle *a* and the aspect ratio *r*, we can always get the horizontal field of view angle β:

$$\beta = 2tan^{-1}\left(r.tan\left(\frac{\alpha}{2}\right)\right)$$

6.3.3.2 Projecting Vertices

As given in Figure 6.14. Given a point (x, y, z), we want to find its projection (x′, y′, d) on the projection plane z = d. Now by using similar triangles and considering x- and y- coordinates separately, we find:

$$\frac{x'}{d} = \frac{x}{z} \Rightarrow x' = \frac{xd}{z} = \frac{xcot(\alpha/2)}{z} = \frac{x}{ztan(\alpha/2)}$$

And

$$\frac{y'}{d} = \frac{y}{z} \Rightarrow y' = \frac{yd}{z} = \frac{ycot(\alpha/2)}{z} = \frac{y}{ztan(\alpha/2)}$$

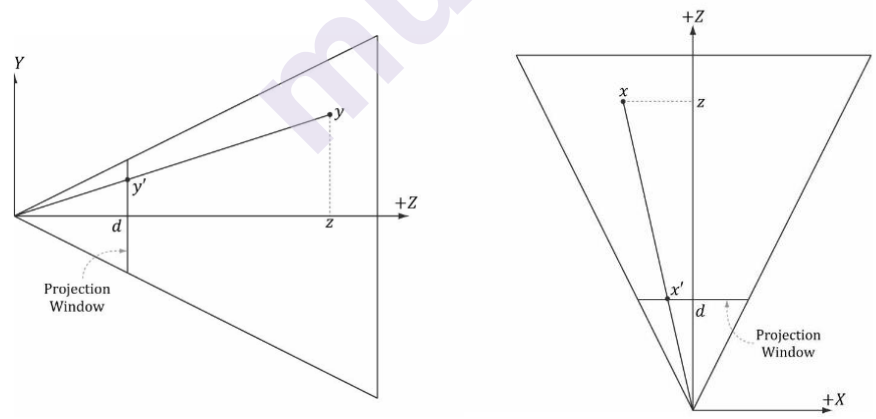


Figure 6.14. Similar triangles.

Observe that a point (x, y, z) is inside the frustum if and only if

$-r \leq x' \leq r$

$-1 \leq y' \leq 1$

$n \leq z \leq f$

6.3.3.3 Normalized Device Coordinates (NDC)

In the previous section we understood that the coordinated of projected points are computer in view space. There, the projection window has a height of 2 and a width of 2r, where 'r' is the aspect ratio. Dimensions being dependent on the aspect ratio is the main problem here. It also means we would need to tell the hardware the aspect ratio that we need, since the hardware will later need to do some operations that involve the dimensions of the projection window (like map it to the back buffer). If we could remove this dependency, it will become more convenient.

One solution is to scale the projected x-coordinate from the interval [–r, r] to [–1, 1] like so:

$-r \leq x' \leq r$

$-1 \leq x' / r \leq 1$

Once this mapping is done, the x-,y-coordinates are said to be normalized device coordinates (NDC) (the z-coordinate has not yet been normalized), and a point (x, y, z) is inside the frustum if and only if

$-r \leq x'/r \leq r$

$-1 \leq y' \leq 1$

$n \leq z \leq f$

This view space to NDC transformation is viewed as a unit conversion. We have the relationship that one NDC unit equals r units in view space (i.e., 1 ndc = r vs) on the x-axis. So given x view space units, we can use this relationship to convert units:

$$x\text{vs}.\frac{1ndc}{r\text{vs}} = \frac{x}{r}ndc$$

It is now easy to modify our projection formulas to give us the projected x- and y-coordinates directly in NDC coordinates:

$$x' = \frac{x}{rz\tan(\alpha / 2)}$$

$$y' = \frac{y}{z\tan(\alpha / 2)} \qquad eq(6.1)$$

Note that in NDC coordinates, the projection window has a height of 2 and a width of 2. Now as the dimensions are fixed, and the hardware need not know the aspect ratio, it is still our responsibility to always supply the projected coordinates in NDC space.

6.3.3.4 Writing the Projection Equation with a Matrix

We will express the projection transformation by a matrix. Equation 6.1, as we have seen is nonlinear, and hence it does not have a matrix representation. We can separate this equation into two parts: 1. a linear part and 2. a nonlinear part. Nonlinear part of the equation is the divided by z. We will see in the next section, how to normalize the z-coordinate; this means we will not have the original z-coordinate for the division operation.

We must save the input z-coordinate before it is transformed; for that, we take the advantage of homogeneous coordinates, and copy the input z-coordinate to the output w-coordinate. As per matrix multiplication, this is done by setting entry [2][3] = 1 and entry [3][3] = 0 (zero-based indices). Our projection matrix looks like this:

$$P = \begin{bmatrix} \frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix}$$

Here we have placed constants A and B into the matrix; they are used to transform the input z-coordinate into the normalized range. Multiplying an arbitrary point (x, y, z, 1) by this matrix gives:

$$[x,y,z,1] \begin{bmatrix} \frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{bmatrix} = \left[\frac{x}{r\tan(\alpha/2)}, \frac{y}{\tan(\alpha/2)}, Az + B, z\right]$$

(eq 6.2)

After this multiplication operation the projection matrix (the linear part), we complete the transformation by dividing each coordinate by w = z (the nonlinear part):

$$\left[\frac{x}{r\tan(\alpha/2)}, \frac{y}{\tan(\alpha/2)}, Az + B, z\right] \xrightarrow{divide\ by\ w} \left[\frac{x}{rz\tan(\alpha/2)}, \frac{y}{\tan(\alpha/2)}, A + B/z, 1\right] \qquad (eq.6.3)$$

There may be a possibility of divide by zero; but, the near plane should be greater than zero, so such a point would be clipped (we will see clipping later). This division by w is called the perspective divide or homogeneous divide. We see that the projected x- and y-coordinates agree with Equation 6.1.

6.3.3.5 Normalized Depth Value

We can discard the original 3D z-coordinate, because all the projected points now placed on the 2D projection window, which is used to forms the 2D image seen by us. For the depth buffering algorithm we need 3D depth information. Just like Direct3D wants the projected x- and y-coordinates in

a normalized range, Direct3D wants the depth coordinates in the normalized range [0, 1]. g(z), which is one order preserving function must be constructed that maps the interval [n, f] onto [0, 1]. To preserve function order, if z1, z2 ∈ [n, f] and z1 < z2, then g(z1) < g (z2); although the depth values have been transformed, the relative depth relationships remain intact, hence we can still correctly compare depths in the normalized interval, which is we actually want for the depth buffering algorithm. Mapping [n, f] onto [0, 1] can be done with two operations those are: scaling and translation. We see from Equation 6.3 that the z-coordinate undergoes the transformation:

$$g(z) = A + \frac{B}{z}$$

Based on the given constraints we need to choose A and B, the conditions are: Condition 1: g(n) = A + B/n = 0 (the near plane gets mapped to zero)Condition 2: g(f) = A + B/f = 1 (the far plane gets mapped to one) When we solve condition 1 for B, it yields: B = –An. Substituting this into condition 2 and solving for A gives:

$$A + \frac{-An}{f} = 1$$

$$\frac{Af - An}{f} = 1$$

$$Af - An = f$$

$$A = \frac{f}{f - n}$$

Therefore,

$$g(z) = \frac{f}{f - n} - \frac{nf}{(f - n)z}$$

A graph of g which is given in Figure 6.15 shows it is strictly increasing (i.e order preserving) and nonlinear. Most of the given range is used up by the depth values of near plane. The majority of the depth values get mapped to a small subset of the range. It may lead to depth buffer precision problems. We can make the near and far planes as close as possible to eliminate the precision problems. Now that we have solved for A and B, we can state the full perspective projection matrix:

$$P = \begin{bmatrix} \frac{1}{r\tan(\alpha/2)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\alpha/2)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{bmatrix}$$

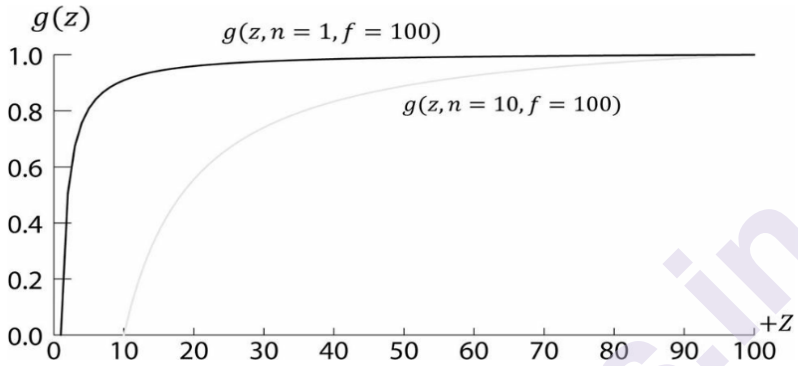


Figure 6.15. Graph of g(z) for different near planes.

Geometry is homogeneous clip or projection space after multiplying by the projection matrix. Then after the perspective divide, the geometry is said to be in normalized device coordinates (NDC).

6.3.3.6 XMMatrixPerspectiveFovLH

The following XNA math function is used for building the perspective projection matrix:

XMMATRIX XMMatrixPerspectiveFovLH(// returns projection matrix
FLOAT FovAngleY, // vertical field of view angle in radians
FLOAT AspectRatio, // aspect ratio = width / height
FLOAT NearZ, // distance to near plane
FLOAT FarZ); // distance to far plane

The code below illustrates how to use **D3DXMatrixPerspectiveFovLH**. Here, we specify a 45° vertical field of view, a near plane at z = 1, and a far plane at z = 1000 (these lengths are in view space).

XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathX::Pi, AspectRatio(), 1.0f, 1000.0f);

The aspect ratio is taken to match our window aspect ratio:

```
float                                   D3DApp::AspectRatio()const
{

return static_cast<float>(mClientWidth) / mClientHeight;

}
```

6.4 THE TESSELLATION STAGE (TS)

Tessellation is one of the important stages in the pipeline. Tessellation means to subdivide the triangles (basic components in image) of a mesh to add new triangles. Newly created triangles can then be offset into new positions to create the fine mesh details as shown in Figure 6.16.

Tessellation provides various benefits as:

1. It helps us to implement a level-of-detail (LOD) mechanism, the triangles near the camera are tessellated to add more detail which leads to a clearer picture, and triangles far away from the camera are not tessellated, which saves extra usage of resources. In this way, we only use more triangles where the extra detail will be noticed.

2. We can keep a simpler low-poly mesh (low-poly means low triangle count in the mesh) in memory, and tessellation can add the extra triangles on the fly, thus saving memory.

3. We can perform operations like animation and physics on a simpler low-poly mesh, and only use the tessellated high-poly mesh for rendering, which helps in producing faster performance.

Tessellation stages are new introduction to Direct3D 11, it mainly provides a way to tessellate geometry on the GPU. Before it's introduction, we needed to perform tessellation activities by using CPU, and then the new tessellated geometry would have to be uploaded back to the GPU for rendering. Uploading new geometry from CPU memory to GPU memory is slow for efficiency, and it also burdens the CPU with computing the tessellation. Hence at that time, tessellation methods have not been very popular for real-time graphics prior to Direct3D 11. Direct3D 11 provides an API to do tessellation operations completely in hardware with a Direct3D 11 capable video card. It makes tessellation an easy to use technique. Tessellation is optional, you can use it if the application demands.

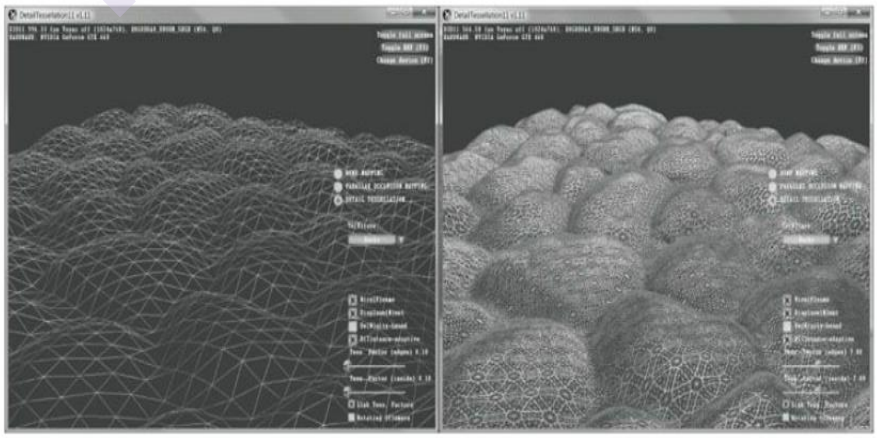


Figure 6.16. The left image shows the original mesh. The right image shows the mesh after tessellation

6.5 GEOMETRY SHADER STAGE (GS)

Like Tessellation, Geometry Shader stage is also optional. It takes entire primitives as an input. For example, consider if we were drawing triangle lists, then the input to the geometry shader will be the three vertices defining the triangle. These vertices are already processed through the Vertex Shader stage. Mainly to create or destroy the geometry, this stage is useful.

Consider the input primitive can be expanded into one or more other primitives, or the geometry shader can choose not to output a primitive, this output depends on some condition that we can assign in Geometry Shader stage.

Geometry Shader is in contrast to a vertex shader, because Vertex Shader cannot create vertices: it inputs one vertex and outputs one vertex. By using Geometry Shader we can convert a point into a quad (ex. square) or a line into the quad. We also notice the "stream-out" arrow from Figure 6.1 (pipeline). Means, the geometry shader can stream-out vertex data into a buffer in memory, which can later be drawn.

6.6 CLIPPING

Sometimes the geometry falls outside of the viewing frustum (can not be viewed) which must be discarded, and geometry that partially intersects the boundary of the frustum must be clipped, in order to preserve only interior part of it and discard the external part; see Figure 6.17 for the idea illustrated in 2D.

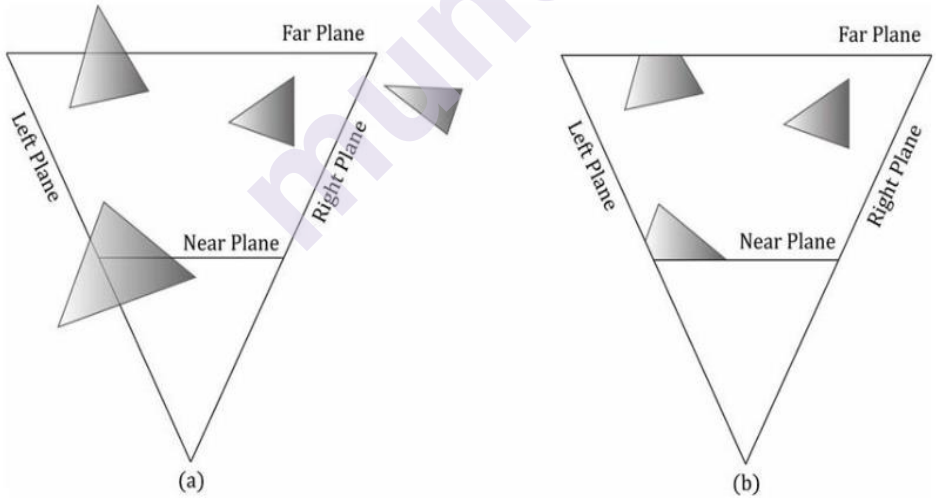


Figure 6.17 (a) Before Clipping (b) After Clipping

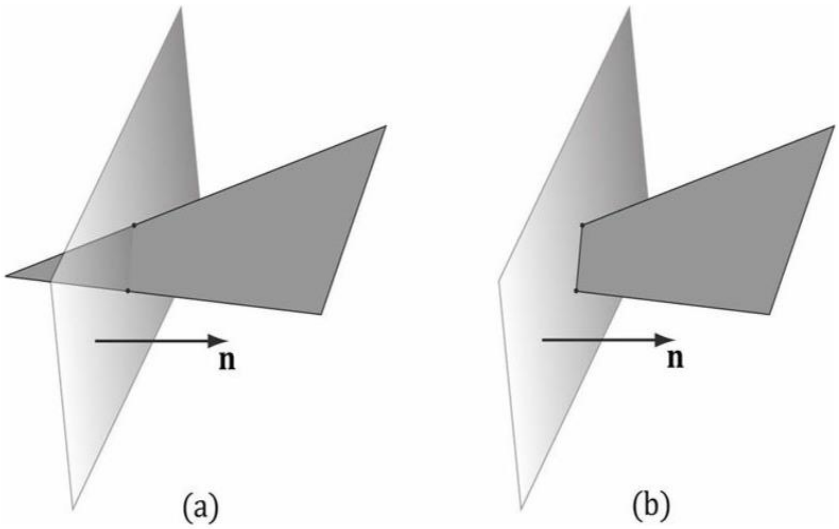


Figure 6.18 (a) Clipping a triangle against a plane. (b) The clipped triangle.

The frustum is a region bounded by six different planes: the top, bottom, left, right, near, and far planes. To perform clipping operation against any polygon on the frustum, we clip it against each frustum plane one by one. When performing clipping operation on frustum (Figure 6.18), the part of the polygon in the positive half space of the plane is kept, and the part in the negative half space of the polygon is discarded. Remember, clipping a convex polygon against a plane will always result in a convex polygon.

Clipping basically amounts to finding the intersection points between the plane and polygon edges, and then ordering the vertices to form the new clipped polygon. Blinn describes how clipping can be done in 4D homogeneous space as shown in Figure 6.19. After the perspective divide is performed, points $\left(\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1\right)$ inside the view frustum are in normalized device coordinates and bounded as follows:

$-1 \leq x/w \leq 1$

$-1 \leq y/w \leq 1$

$0 \leq z/w \leq 1$

Hence, in homogeneous clip space, before the divide, 4D points (*x, y, z, w*) inside the frustum are bounded as follows:

$-w \leq x \leq w$

$-w \leq y \leq w$

$0 \leq z \leq w$

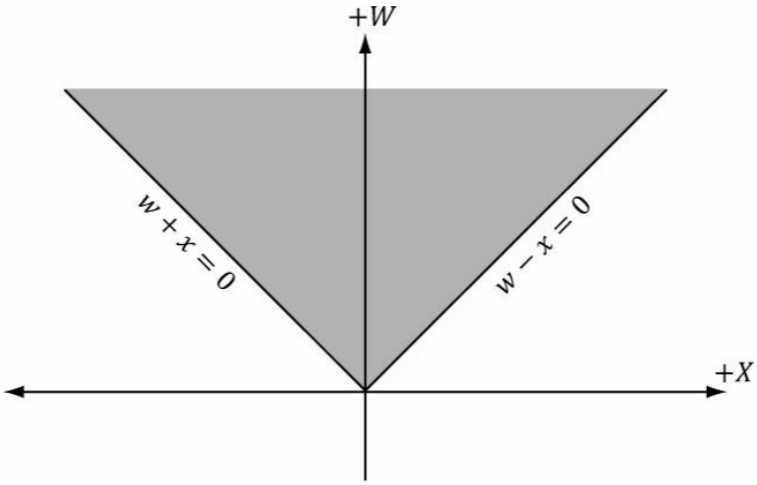


Figure 6.19. The frustum boundaries in the xw-plane in homogeneous clip space.

The points are bounded by the simple 4D planes:

Left: $w = -x$
Right: $w = x$

Bottom: $w = -y$

Top: $w = y$

Near: $z = 0$
Far: $z = w$

After knowing frustum plane equations in homogeneous space, we can apply a clipping algorithm (like Sutherland-Hodgeman).

6.7 THE RASTERIZATION STAGE

Another important stage in the pipeline is the Rasterization stage. Its main job is to compute pixel colors from the projected 3D triangles.

6.7.1 Viewport Transform

After clipping operation, the hardware can do the perspective divide step for transforming from homogeneous clip space to normalized device coordinates (NDC). The 2D x- and y- coordinates forming the 2D image are transformed to a rectangle on the back buffer called the viewport after vertices are in NDC space.

Once this operation is performed, the x- and y-coordinates are in units of pixels. The viewport transformation does not modify the z-coordinate, as it is used for depth buffering, although it can by modifying the MinDepth and MaxDepth values of the D3D11_VIEWPORT structure. The range of MinDepth and MaxDepth values must be between 0 and 1.

6.7.2 Backface Culling

In a triangle, to distinguish between the two sides of it we use the following convention. In the case where triangle vertices are ordered v_0, v_1, v_2 then we compute the triangle normal n by using formula like:

$e_0 = v_1 - v_0$

$e_1 = v_2 - v_0$

$$n = \frac{e_0 X\ e_1}{|\ e_0 X\ e_1\ |}$$

The side the normal vector emanates from is the front side and the other side is the back side. Figure 6.20 illustrates this. If front side of the triangle is visible to the user then we can say that, the triangle is front facing, and we say a triangle is back-facing if the viewer sees the back side of a triangle.

With the perspective of Figure 6.20, the left triangle is front-facing while the right triangle is back-facing. Notice, from our perspective, the left triangle is ordered in clockwise direction while the right triangle is ordered in counterclockwise direction.

It is not a coincidence: because with the convention we have chosen (i.e., the way we compute the triangle normal), a triangle which is ordered clockwise (with respect to that viewer) is front-facing, and a triangle which is ordered counterclockwise (with respect to that viewer) is back-facing. Most objects in 3D worlds are the enclosed solid objects. So, suppose we are constructing the triangles for each object in such a way that the normals are always aimed outward. Then, the camera won't see the back-facing triangles of a solid object because the front-facing triangles occlude the back-facing triangles; as Figure 6.21 illustrates this in 2D and Figure 6.22 in 3D. Because the front-facing triangles occlude the back-facing triangles, it makes no sense to draw them. Backface culling is a term which refers to the process of discarding back-facing triangles from the pipeline. This is helpful in reducing the amount of triangle processing by almost half.

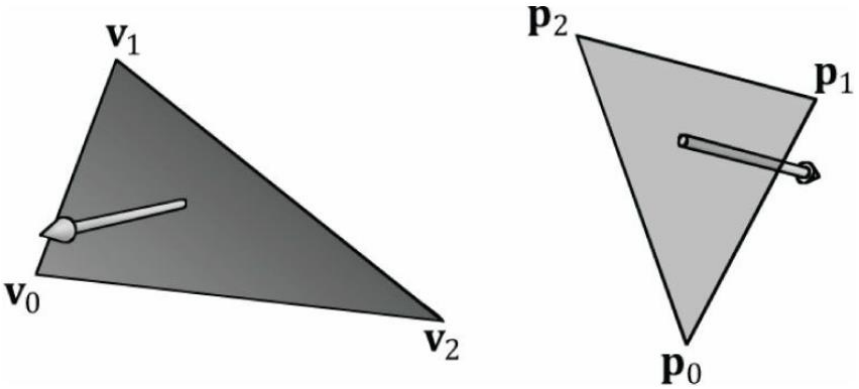


Figure 6.20. The left triangle is front-facing from our viewpoint, and the right triangle is back facing from our view point

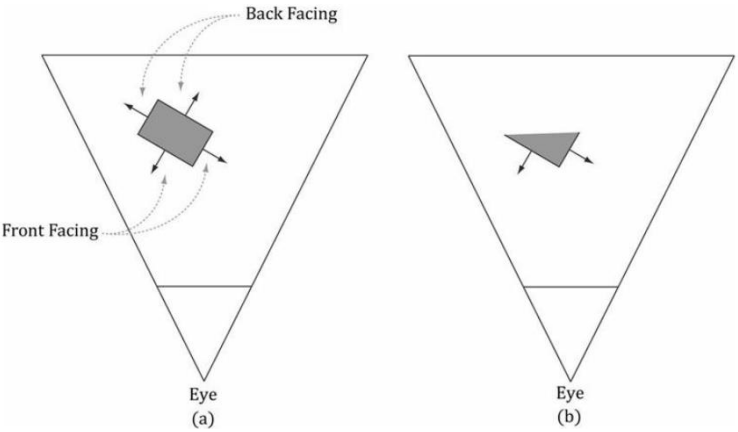


Figure 6.21 (a) A solid object with front-facing and back-facing triangle. (b) The scene after culling the back facing triangles.

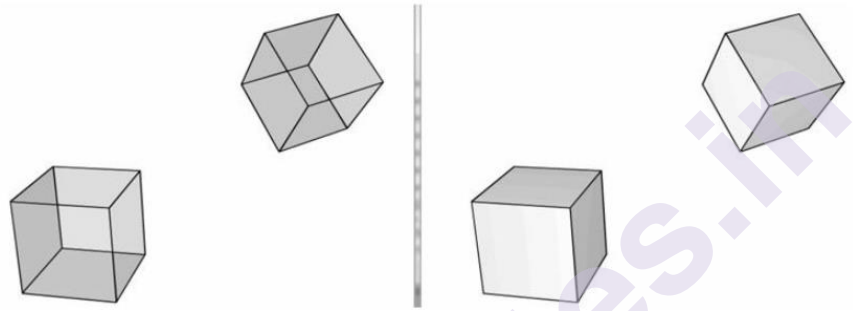


Figure 6.22. (Left) We draw the cubes with transperancy so that you can see all six sides. (Right) We draw the cubes as solid blocks.

Direct3D treats triangles with a clockwise winding order (with respect to the viewer) by default as front-facing, and triangles with a counterclockwise winding order (with respect to the viewer) as back-facing. This whole convention can be reversed with a Direct3D render state setting.

6.7.3 Vertex Attribute Interpolation

We have seen how to define a triangle by specifying its vertices. Along with position, we can attach other attributes to vertices such as colors, normal vectors, and texture coordinates as well. After the viewport transform, these attributes need to be interpolated (the operation we will see later) for each pixel covering the triangle.

Along with vertex attributes, vertex depth values need to get interpolated so that each pixel has a depth value for the depth buffering algorithm. The vertex attributes are interpolated too in screen space in such a way that the attributes are interpolated linearly across the triangle in 3D space as shown in Figure 6.23; which requires the so-called *perspective correct interpolation*. The interpolation allows us to use the vertex values to compute values for the interior pixels. We need not worry about the mathematical details of perspective correct attribute interpolation because the hardware does it for us. Figure 6.24 gives the basic idea about the technique.

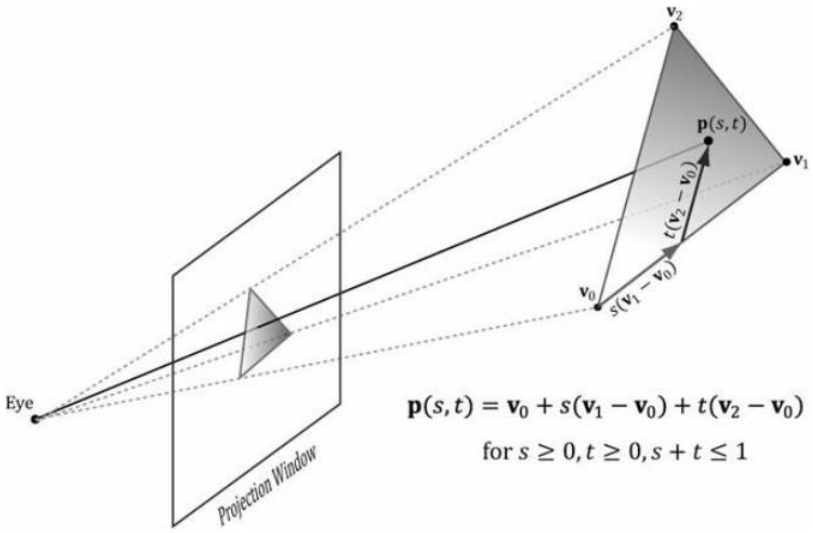


Figure 6.23. An attribute value p(s,t) on a triangle can be obtained by linearly interpolating between the attribute values at the vertices of the triangle

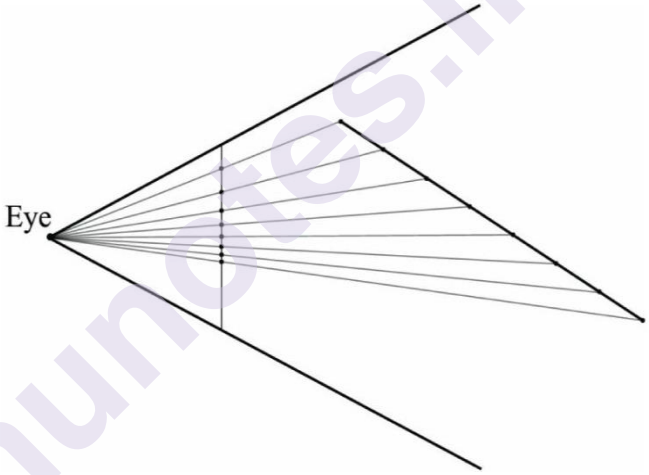


Figure 6.24. A 3D line is being projected onto the projection window (the projection is a 2D line in screen space). We see that taking uniform step sizes along the 3D line corresponds to taking nonuniform step sizes in 2D screen space.

6.8 PIXEL SHADER STAGE (PS)

The Pixel Shaders are programs that we write but are executed by GPU as we have seen for Vertex Shaders. A pixel shader is executed for every pixel fragment and uses the interpolated vertex attributes as input to compute a color.

The same way we have seen in VS for each vertex. Main use of pixel shader is to deal with pixel colors. It can be as simple as returning a constant color, to doing more complex things like per-pixel lighting, reflections, and shadowing effects.

6.9 OUTPUT MERGER STAGE (OM)

As the name indicates this stage merges the final outputs. Hence, after pixel fragments have been generated by the pixel shader stage, they move onto the output merger (OM) stage of the rendering pipeline. Here, some pixel fragments may be rejected (like, from the depth or stencil buffer tests).

These pixel fragments which are not rejected are written to the back buffer. Blending (which we will address later) is also done in this stage, where a pixel may be blended with the pixel which is currently on the back buffer instead of overriding it completely.

6.10 UNDERSTANDING MESHES OR OBJECTS, TEXTURING, LIGHTING, BLENDING

6.10.1 Understanding Meshes or Objects

Mesh is an important aspect in computer graphics. A polygon mesh is the collection of vertices, edges, and faces (the components we already know) which is helpful in making attractive and realistic 2D and 3D object. The polygon mesh contains the shape and contour for every 3D character and/or object. Further this can be used in animated films, games, advertisements etc.

We can understand the polygon mesh in an easy manner. Each vertex in that contains the x, y and z coordinate information. Then the surface information is contained in every face for that polygon. Which is further used to render the scene using rendering engine and to calculate lighting and shadows.

We model the polygon mesh which is used to approximate the 3D surface with lines and polygons. Blender, Maya are some of the popular programs that are used for creating polygon meshes. For modeling, texturing the animated objects these tools mostly used.

As we know the 3D objects are solid the polygon mesh are not. Most of the meshes that we create are rendered as the polygonal quads; then they are split into the triangles by computer.

There are two faces for every quad the front and back face. The surface angle is calculated with front face and back face is hidden from the camera.

There are few limitations in polygon meshes, curved surfaces are difficult to approximate with a series of lines. Small objects like hair and liquid are difficult to simulate using polygon meshes.

All the animated characters in games and cartoons are made up of meshes. One important property that meshes have is the ability of deformation; which helps them to move, run, twist, etc.

Adding a texture and color on the mesh will bring the character to life. And make it attractive. The modern computer graphics world is made up of polygon meshes.

6.10.2 Texturing

One important aspect in any animation is the Texture any object has, which helps the animation to be more attractive and real. The Texture mapping technique is the one that allows us to map image data onto a triangle, hence enabling us to increase the details and realism of our scene in significant manner. For instance, we can build a cube and turn it into a crate by mapping a crate texture on each side as you can see in Figure 6.25.

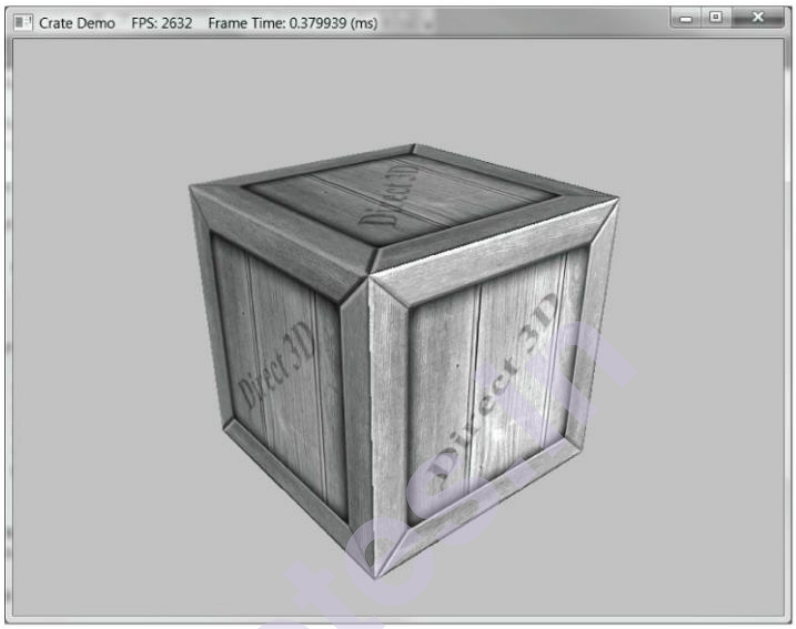


Figure 6.25. The Crate demo creates a cube with a crate texture

Texture And Resource Recap

We already know that, the depth buffer and back buffer are 2D texture objects represented by the **ID3D11Texture2D** interface. In the first section we will review much of the material on textures. A 2D texture, as we know, is a matrix of data elements. We use 2D textures to store 2D image data, where the color of a pixel is stored in each of the element. But, this is not the only usage of textures; consider, in an advanced technique called as normal mapping, each element in the texture stores a 3D vector instead of a color.

Although it is common for textures to store image data, they are very general purpose than that. Consider a 1D texture (**ID3D11Texture1D**) is like a 1D array of data elements, and a 3D texture (**ID3D11Texture3D**) is like a 3D array of data elements. Here, the 1D, 2D, and 3D texture interfaces all inherit from **ID3D11Resource**. We will see later, how textures are more than just arrays of data; also they can have mipmap levels, and how the GPU can do special operations on textures, like to apply filters and multisampling.

Textures are not arbitrary chunks of data; they can only store certain kinds of data formats, which are described by the DXGI_FORMAT enumerated type. Some example are:

DXGI_FORMAT_R32G32B32_FLOAT	Every element has three 32-bit floating-point components.
DXGI_FORMAT_R16G16B16A16_UNORM	Every element has four 16-bit components mapped to the [0, 1] range.
DXGI_FORMAT_R32G32_UINT	Every element has two 32-bit unsigned integer components.
DXGI_FORMAT_R8G8B8A8_UNORM	Every element has four 8-bit unsigned components mapped to the [0, 1] range.
DXGI_FORMAT_R8G8B8A8_SNORM	Every element has four 8-bit signed components mapped to the [–1, 1] range.
DXGI_FORMAT_R8G8B8A8_SINT	Every element has four 8-bit signed integer components mapped to the [–128, 127] range.
DXGI_FORMAT_R8G8B8A8_UINT	Every element has four 8-bit unsigned integer components mapped to the [0, 255] range.

Remember that, the R, G, B, A letters are used to stand for red, green, blue, and alpha, respectively. However, as we said earlier, textures need not store color information; for example, the format **DXGI_FORMAT_R32G32B32_FLOAT.**

This format has three floating-point components and can therefore store a 3D vector with floating-point coordinates. We can use typeless formats too, in those we just reserve memory and then specify how to reinterpret the data later. When the texture is bound to the rendering pipeline; consider example, the following typeless format reserves elements with four 8-bit components, but does not specify the data type (e.g., the general data types as integer, floating-point, unsigned integer):

DXGI_FORMAT_R8G8B8A8_TYPELESS

One texture can be bound to different stages of the rendering pipeline; a common example is to use a texture as a render target (for instance,

Direct3D draws into the texture) and as a shader resource as well (like, the texture will be sampled in a shader). This resource which is created for two purposes is given by following binding flags:

D3D11_BIND_RENDER_TARGET | D3D11_BIND_SHADER_RESOURCE

It indicates the two pipeline stages the texture will be bound to. The resources are not directly bound to a pipeline stage; but, their associated resource views are bound to different pipeline stages. If we are using textures in any way, Direct3D requires that we create a resource view of that texture at the time of initialization.

It is done for efficiency, as the SDK documentation points out: "This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time." For the example of using a texture as a render target and shader resource, consider creation of two views: a render target view (**ID3D11RenderTargetView**) and a shader resource view (**ID3D11ShaderResourceView**). Two things are done with Resource Views: First, they tell Direct3D how the resource will be used, and second, if the resource format was specified as typeless at creation time, then we must now state the type when creating a view. Hence, with typeless formats, it is possible for the elements of a texture to be viewed as floating-point values in one pipeline stage and as integers in another; this essentially amounts to a reinterpret cast of the data.

We should only create a typeless resource if we really need it; else, create a fully typed resource. To create specific view for a resource, the resource should be created with that specific bind flag. Consider for example, if the resource was not created with the **D3D11_BIND_SHADER_RESOURCE** bind flag (which indicates the texture will be bound to the pipeline as a depth/stencil buffer), then we cannot create an **ID3D11ShaderResourceView** to that resource.

If we will try, we will get an error ike the following: **D3D11: ERROR: ID3D11Device::CreateShaderResourceView: A ShaderResourceView cannot be created of a Resource that did not specify the D3D11_BIND_SHADER_RESOURCE BindFlag.**

6.10.3 Texture Coordinates

There are two texture coordinates used in Direct3D a u-axis that runs horizontally to the image and a v-axis that runs vertically to the image. These coordinates, (u, v) such that $0 \leq u, v \leq 1$, identify *texel*, an element on the texture. Notice that, the v-axis is positive in the "down" direction consider Figure 6.26.

Notice the normalized coordinate interval, [0, 1], which is used because it gives a range dependent on dimensions for Direct3D to work with; for example, range like (0.5, 0.5) always specifies the middle texel independent on if the actual texture dimensions is 256 × 256, 512 × 1024, or 2048 × 2048 in pixels. Similarly, (0.25, 0.75) identifies the texel a quarter of the total

width in the horizontal direction, and three-quarters of the total height in the vertical direction. We will consider, the texture coordinates always in the range [0, 1], but later we explain what can happen when you go outside this range.

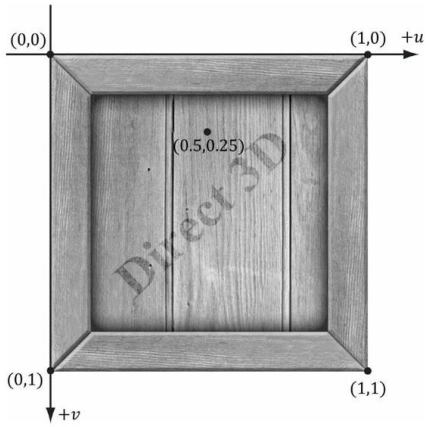


Figure 6.26. The texture coordinate system, sometimes called texture space

We will define a corresponding triangle on texture for each 3D triangle, that is to be mapped onto the 3D triangle as shown in Figure 6.27. Let p0, p1, and p2 be the vertices of a 3D triangle with respective texture coordinates q0, q1, and q2. Consider any arbitrary point as (x, y, z) on the 3D triangle, the texture coordinates (u, v) are found by the linear interpolation operation on the vertex texture coordinates across the 3D triangle by the same s, t parameters; that is, if

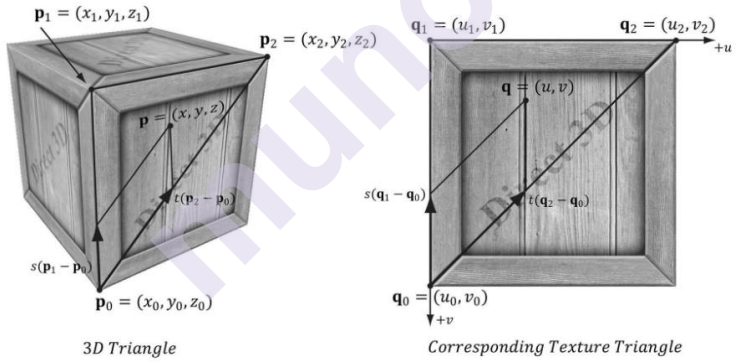


Figure 6.27. On the left is a triangle in 3D space, and on the right we define a 2D triangle on the texture that is going to be mapped onto the 3D triangle.

$(x, y, z) = p = p_0 + s\,(p_1 - p_0) + t\,(p_2 - p_0)$ for $s \geq 0$, $t \geq 0$, $s + t \leq 1$
then,
$(u, v) = q = q_0 + s(q_1 - q_0) + t(q_2 - q_0)$

We can see, every point on the triangle has a corresponding texture coordinate assigned to it. For implementation, we will modify our vertex structure and add a pair of texture coordinates which will help to identify a point on the texture.

Here every 3D vertex has a corresponding 2D texture vertex. Hence, every 3D triangle defined by three vertices also defines a corresponding 2D triangle in texture space.

```
// Basic 32-byte vertex structure given by a code:
struct Basic32
{
XMFLOAT3 Pos;
XMFLOAT3 Normal;
XMFLOAT2 Tex;
};
const D3D11_INPUT_ELEMENT_DESC InputLayoutDesc::Basic32[3] =
{
{"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0, D3D11_INPUT_PER_VERTEX_DATA, 0},
{"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 12, D3D11_INPUT_PER_VERTEX_DATA, 0},
{"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 24, D3D11_INPUT_PER_VERTEX_DATA, 0}
};
```

If the 2D triangle is much different than the 3D one then you can create the 'odd' texture mappings. Hence, when the 2D texture is mapped onto the 3D triangle, a lot of stretching and distortion occurs and the result will not look good. For example, when we map an acute angled triangle to the right angled triangle, it requires stretching. In general, texture distortion should be minimized, unless the texture artist desires the distortion look.

See in Figure 6.27, we are mapping whole texture image onto each face of the cube. This is not required. We can map only a part/subset of a texture onto geometry. We can play several unrelated images on one big texture map, and use it for several different objects as shown in Figure 6.28). The texture coordinates are what will determine what part of the texture gets mapped on the triangles.

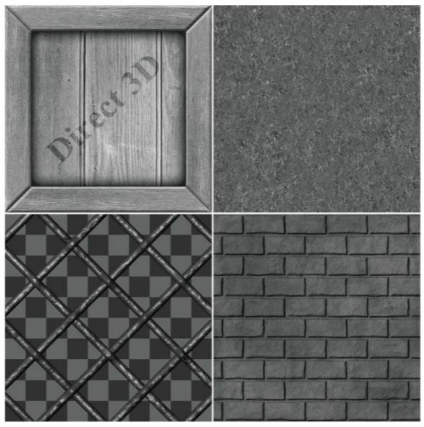


Figure 6.28. A texture atlas storing four subtextures on one large texture. The texture coordinates for each vertex are set so that the desired part of the texture gets mapped onto the geometry.

6.10.4 Creating And Enabling A Texture

We usually read texture data from an image file which is stored on disk and loaded into an **ID3D11Texture2D** object. Although, texture resources are not bound directly to the rendering pipeline; you create a shader resource view (**ID3D11ShaderResourceView**) to the texture, and then bind the view to the pipeline. Two steps are to be taken to achieve this as below:

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 the 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
);
```

Here,

1. **pDevice:** Is a Pointer to the D3D device to create the texture with.

2. **pSrcFile**: Is the Filename of the image to load.

3. **pLoadInfo**: Is an Optional image info; specify null to use the information from the source image.

4. **pPump**: Is used to spawn a new thread for loading the resource. To load the resource in the main thread, specify null.

5. **ppShaderResourceView:** It returns a pointer to the created shader resource view to the texture loaded from file.

6. **pHResult:** It specify null if null was specified for pPump.

Any format can be loaded with the given function as: BMP, JPG, PNG, DDS, TIFF, GIF, and WMP. We can refer to a texture and its corresponding shader resource view as interchangeable. For example, we may say we are binding the texture to the pipeline, even though we are really binding its view.

For example consider, to create a texture from an image called WoodCreate01.dds, we would write the following:

```
ID3D11ShaderResourceView* mDiffuseMapSRV;
HR(D3DX11CreateShaderResourceViewFromFile(md3dDevice,
L"WoodCrate01.dds", 0, 0, &mDiffuseMapSRV, 0 ));
```

Once we load this texture, we need to set it to an effect variable so that it can be used in a pixel shader. A 2D texture object in an .fx file is represented by the Texture2D type; for example, we declare a texture variable in an effect file like so:

```
// Nonnumeric values cannot be added to a cbuffer.
Texture2D gDiffuseMap;
```

As given in comment, texture objects are placed outside of constant buffers. We can obtain a pointer to an effect's Texture2D object (which is a shader resource variable) from our C+ + application code as follows:

```
ID3DX11EffectShaderResourceVariable* DiffuseMap;
fxDiffuseMap = mFX->GetVariableByName("gDiffuseMap")-
>AsShaderResource();
```

Once we have obtained a pointer to an effect's Texture2D object, we can update it through the C+ + interface like so:

```
// Set the C++ texture resource view to the effect texture variable.
fxDiffuseMap->SetResource(mDiffuseMapSRV);
```

As with other effect variables, if we need to change them between draw calls, we must call Apply:

```
// set crate texture
fxDiffuseMap->SetResource(mCrateMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawCrate();
// set grass texture
fxDiffuseMap->SetResource(mGrassMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawGrass();
// set brick texture
fxDiffuseMap->SetResource(mBrickMapSRV);
pass->Apply(0, md3dImmediateContext);
DrawBricks();
```

The texture atlases can improve performance because it can lead to drawing more geometry with one draw call. Suppose we used the texture atlas as given in Figure 6.28 that contains the crate, grass, and brick textures. Then adjust the texture coordinates for each object to its corresponding subtexture, we could draw the geometry in one draw call (assuming no other parameters needed to be changed per object):

```
// set texture atlas

fxDiffuseMap->SetResource(mAtlasSRV);
pass->Apply(0, md3dImmediateContext);
DrawCrateGrassAndBricks();
```

There is overhead to draw calls, so it is desirable to minimize them with techniques like this.

A texture resource can actually be used by any shader (vertex, geometry, or pixel shader). For now, we will just be using them in pixel shaders. We

already know that, textures are essentially special arrays, so it is not hard to imagine that array data could be useful in vertex and geometry shader programs, too.

6.11 LIGHTING

See Figure 6.29, which is helpful to understand the significance of lighting the objects. The left side of figure shows an unlit (without light) sphere, and on the right hand side, we have a lit sphere. The left sphere looks very flat, like a circle in 2D. The right side sphere looks like an actual sphere in 3D. The lighting and shading effects aid in our perception of the solid form and volume of any object. Our visual understanding and perception of the world depends on light, which falls on material, and the light itself in the scene, and physically accurate lighting models play important role in much of the problem of generating photorealistic scenes.

The expensiveness of model is of course dependent on the accuracy of model, the more accurate will be more expensive; we must keep a balance between realism and speed. To understand this, consider the 3D special FX scenes for films that can be much more complex and utilize very realistic lighting models than a game because in films we use pre-rendered frame, whereas in games, we render frames on the go, so films can afford to take hours or days to process a frame. In games, the frames need to be drawn at a rate of at least 30 frames per second.

6.11.1 Light And Material Interaction

There is no need to specify vertex colors while using lighting, instead we specify the materials and lights, after that we apply a lighting equation by using which machine can compute vertex colors based on the interaction of light and material. This technique helps us to attain more realistically colored objects. (you can compare Figure 6.29a and 6.29b again).

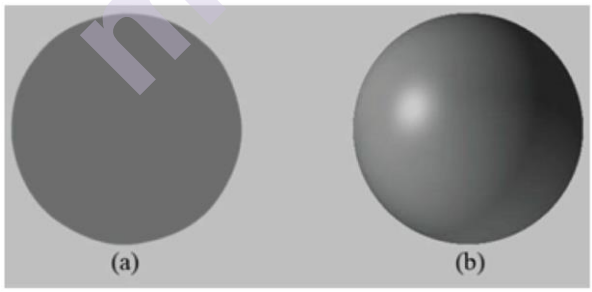


Figure 6.29. (a) An unlit sphere looks 2D. (b) A lit sphere looks 3D.

Materials are the properties that determine how light interacts with a surface of an object. For example, the material of a surface is made up of the parameters as the colors of light a surface reflects and absorbs, and also the reflectivity, transparency, and shininess of surface. In the model that we will consider in this text, a light source can emit different intensities of red, green, and blue light; by using the combination of them, we can simulate many light colors.

When the source emits the light which collides with any object, part of light may be absorbed and part may be reflected; if object is transparent then light may pass through it (ex. glass). When the light reflects travels along the new path and may collide with other objects, and then again may partly absorbed and reflected.

This partial absorption and reflectance means a light ray may strike many objects before it is fully absorbed. Some light rays eventually travel into the eye (refer Figure 6.30 to understand) and strike the light receptor cells (known as cones and rods) on the retina in the eyes.

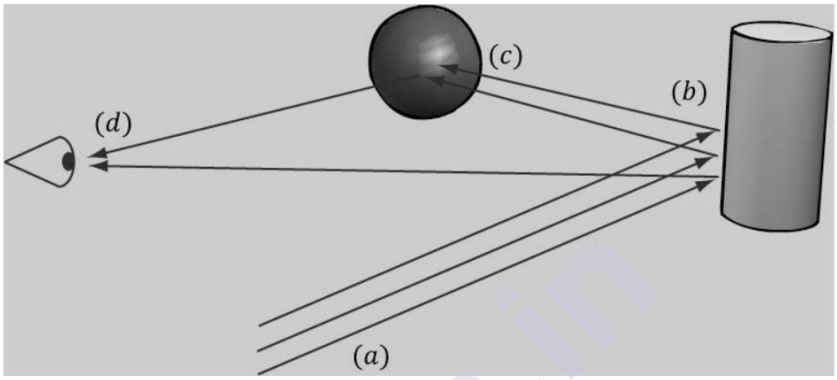


Figure 6.30. (a) Flux of incoming white light. (b) The light strikes the cylinder and some rays are absorbed and other rays are scatted toward the eye and sphere. (c) The light reflecting off the cylinder toward the sphere is absorbed or reflected again and travels into the eye. (d) The eye receives incoming light that determines what the eye sees.

The trichromatic theory states, our retina contains three kinds of light receptors, every one sensitive to red, green, and blue light. The incoming RGB (Red, Green, and Blue) light stimulates the respective light receptor with variable intensity, which is based on the strength a light carries. After this stimulation, some neural impulse is sent down the optic nerve connected to brain, and brain generates the image based on the stimulus.

See Figure 6.30 again and suppose cylinder material reflects 75% red light, 75% green light, and absorbs the remaining light, and the sphere material reflects 25% red light and absorbs the remaining light. Suppose the light source in the scene emits pure white light. When the light rays from source strike the cylinder, the blue light will be completely absorbed and 75% of red and green light is reflected (i.e kind of yellow light). This light is scattered, part of it may travel into eyes and part will fall on sphere. Hence, we will see the cylinder as a semi-bright shade of yellow. The remaining light rays travel toward the sphere and strike on it.

As mentioned the sphere reflects 25% red light and absorbs blue and green completely; hence, the medium-high intensity red light is diluted further and reflected. This red light then travels into our eyes thus the we see the sphere as a dark shade of red. We will adopt local illumination lighting models in this text in particular. Every object is lit independently of another object,

and only the light directly emitted from light sources is taken into account in the lighting process while using this local model.

The Figure 6.31 shows a consequence of this model. The global illumination models, on the other hand, not only considers the direct light from the light sources but also considers the light which is bounced off from other objects in the scene. They are called global because they take every light in the scene into consideration, they are expensive and mainly used to create photorealistic effects in games.

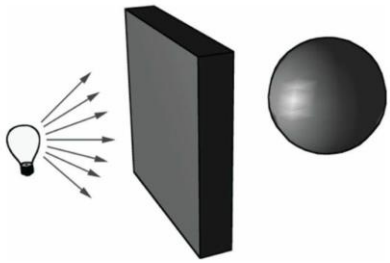


Figure 6.31. Physically, the wall blocks the light rays emitted by the light bulb and the sphere is in the shadow of the wall. However, in a local illumination model, the sphere is lit as if the wall were not there.

6.11.2 Normal Vectors

Normal vector is very important concept to be understood before moving forward. A face normal (unit vector) which describes the direction a polygon is facing; refer Figure 6.32a for this. Surface normal is also the unit vector which is orthogonal to the tangent plane of a point on a surface; refer Figure 6.32b for this.

Direction of a point on a surface is facing is determined by surface normal. Surface normal is required at each point on the triangle mesh surface for lighting calculations so that we can determine the angle at which light strikes the point on the mesh surface. We specify the surface normals only at the vertex points also known as vertex normals. To obtain a surface normal approximation at each point on the surface of a triangle mesh, these vertex normals will be interpolated across the triangle during rasterization (refer Figure 6.33).

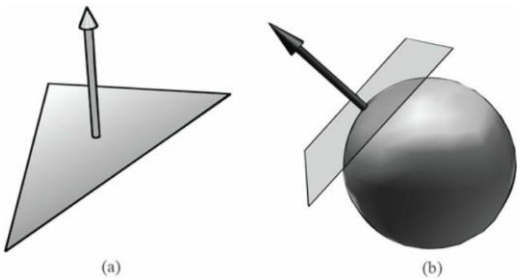


Figure 6.32. (a) The face normal is orthogonal to all points on the face. (b) The surface normal is the vector that is orthogonal to the tangent plane of a point on a surface.

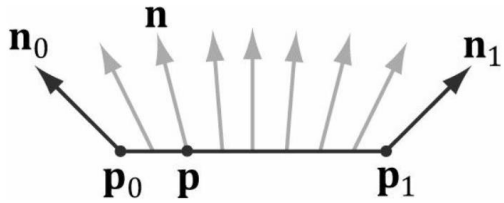


Figure 6.33. The vertex normals n0 and n1 are defined at the segment vertex points p0 and p1. A normal vector n for a point p in the interior of the line segment is found by linearly interpolating (weighted average) between the vertex normals; that is, n = n0 + t (n1 – n0), where t is such that p = p0 + t (p1 – p0). Although we illustrated normal interpolation over a line segment for simplicity, the idea straightforwardly generalizes to interpolating over a 3D triangle.

6.11.2.1 Computing Normal Vectors

For finding face normal of triangle with points $\Delta p_0\ p_1\ p_2$, we will determine the two vectors lying on the triangle's edge as:

$u = p_1 - p_0$
$v = p_2 - p_0$

Then the face normal calculated as:

$$n = \frac{u\ X\ v}{|\ u\ X\ v\ |}$$

The function given below computes face normal for the front side of the triangle from the three vertex points.

```
void ComputeNormal(const D3DXVECTOR3& p0,
const D3DXVECTOR3& p1,
const D3DXVECTOR3& p2,
D3DXVECTOR3& out)
{
D3DXVECTOR3 u = p1 - p0;
D3DXVECTOR3 v = p2 - p0;
D3DXVec3Cross(&out, &u, &v);
D3DXVec3Normalize(&out, &out);
}
```

If the surface would have been differentiable, we could have used calculus to compute normal; but triangle mesh is not differentiable. Hence we use a technique called as vertex normal averaging.

For every polygon in the mesh, which shares a vertex v, the vertex normal n is calculated by averaging the face normal of that polygon. For example, consider Figure 6.34, which shows four polygons in the mesh share the vertex v; thus, the vertex normal for v is given by:

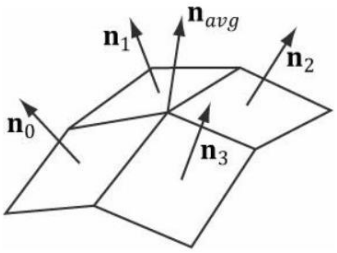


Figure 6.34. The middle vertex is shared by the neighboring four polygons, so we approximate the middle vertex normal by averaging the four polygon face normals.

$$n_{avg} = \frac{n_0 + n_1 + n_2 + n_3}{| n_0 + n_1 + n_2 + n_3 |}$$

To calculate the average, there is no need to divide by 4 in previous example because we normalize the result.

We can construct more sophisticated averaging techniques also. For example, we can use a weighted average where the weights are determined by the areas of the polygons. The code given below shows the implementation of this averaging when vertex and index list of triangle mesh is given:

```
// Input:
// 1. An array of vertices (mVertices). Each vertex has a
// position component (pos) and a normal component (normal).
// 2. An array of indices (mIndices).
// For each triangle in the mesh:
for(UINT i = 0; i < mNumTriangles; ++i)
{
// indices of the ith triangle
UINT i0 = mIndices[i*3+0];
UINT i1 = mIndices[i*3+1];
UINT i2 = mIndices[i*3+2];
// vertices of ith triangle
Vertex v0 = mVertices[i0];
Vertex v1 = mVertices[i1];

Vertex v2 = mVertices[i2];
// compute face normal
Vector3 e0 = v1.pos - v0.pos;
Vector3 e1 = v2.pos - v0.pos;
Vector3 faceNormal = Cross(e0, e1);
// This triangle shares the following three vertices,
// so add this face normal into the average of these
// vertex normals.
mVertices[i0].normal += faceNormal;
mVertices[i1].normal += faceNormal;
mVertices[i2].normal += faceNormal;
}
```

```
// For each vertex v, we have summed the face normals of all
// the triangles that share v, so now we just need to normalize.
for(UINT i = 0; i < mNumVertices; ++i)
mVertices[i].normal = Normalize(&mVertices[i].normal));
```

6.11.2.2 Transforming Normal Vectors

See Figure 6.35a, which has a tangent vector $u = v_1 – v_0$ orthogonal to a normal vector n. If 'A' is considered as a nonuniform scaling transformation, we can see from Figure 7.7b that the transformed tangent vector $u\ A = v_1A – v_0A$ doesn't remain orthogonal to the transformed normal vector nA.

The problem is described as: Given a transformation matrix A that transforms points and vectors (non-normal), we want to find a transformation matrix B that transforms normal vectors such that the transformed tangent vector is orthogonal to the transformed normal vector (i.e., $uA \cdot nB = 0$). For this let us first start with what we have: we know that the normal vector n is orthogonal to the tangent vector u:

$\mathbf{u} \cdot \mathbf{n} =$	Tangent vector orthogonal to normal vector
$\mathbf{u}\mathbf{n}^T = 0$	Rewriting the dot product as a matrix multiplication
$\mathbf{u}(\mathbf{A}\mathbf{A}^{-1})\ \mathbf{n}^T = 0$	Inserting the identity matrix $\mathbf{I} = \mathbf{A}\mathbf{A}^{-1}$
$(\mathbf{u}\mathbf{A})(\mathbf{A}^{-1}\ \mathbf{n}^T) = 0$	Associative property of matrix multiplication
$(\mathbf{u}\mathbf{A})\ ((\mathbf{A}^{-1}\ \mathbf{n}^T)^T)^T = 0$	Transpose property $(\mathbf{A}^T)^T = \mathbf{A}$
$(\mathbf{u}\mathbf{A})\ (\mathbf{n}(\mathbf{A}^{-1})^T)^T = 0$	Transpose property $(\mathbf{AB})^T = \mathbf{B}^T\mathbf{A}^T$
$\mathbf{u}\mathbf{A} \cdot \mathbf{n}(\mathbf{A}^{-1})^T = 0$	Rewriting the matrix multiplication as a dot product
$\mathbf{u}\mathbf{A} \cdot \mathbf{n}\mathbf{B} = 0$	Transformed tangent vector orthogonal to transformed normal vector

Thus $B = (A^{-1})^T$ (i.e the inverse transpose of A) is used to transform normal vectors hence, they will be perpendicular to their associated transformed tangent vector uA.

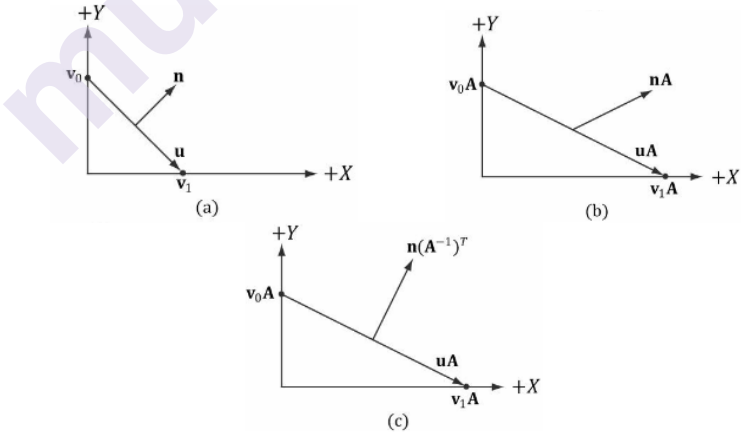


Figure 6.35. (a) The surface normal before transformation. (b) After scaling by 2 units on the x-axis the normal is no longer orthogonal to the surface. (c) The surface normal correctly transformed by the inverse-transpose of the scaling transformation.

We do not need to calculate the inverse transpose because as A does the job here; as if the matrix is orthogonal ($A^T = A–1$), then $B = (A–1)^T = (A^T)^T = A$. To summarize we can say that, use the inverse transpose when

transforming a normal vector by a nonuniform or shear transformation. To compute the inverse-transpose we will use the helper function in MathHelper.h:

```
static XMMATRIX InverseTranspose(CXMMATRIX M)

{
XMMATRIX A = M;
A.r[3] = XMVectorSet(0.0f, 0.0f, 0.0f, 1.0f);
XMVECTOR det = XMMatrixDeterminant(A);
return XMMatrixTranspose(XMMatrixInverse(&det, A));
}
```

Every matrix translation will be cleared now as we are using inverse-transpose for transforming vectors and as translations are only applied to points. If we set w=0 for vectors (using homogeneous coordinates), it prevents vectors from being modified by translations. Hence, there is no need to zero out the matrix translation.

Concatenation of an inverse-transpose and a matrix doesn't contain nonuniform scaling will cause a problem., say the view matrix $(A^{-1})^T$ V, the transposed translation in the 4th column of $(A–1)^T$ "leaks" into the product matrix causing errors. So, we zero out the translation as a precaution to avoid this kind of an error. To achieve things in proper way is to transform the normal by $((AV)^{-1})^T$. Observe the example below showing a scaling and translation matrix, and how the inverse-transpose looks with a 4th column not $[0, 0, 0, 1]^T$:

$$A = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0.5 & 0 & 0 \\ 0 & 0 & 0.5 & 0 \\ 1 & 1 & 1 & 1 \end{bmatrix}$$

$$(A^{-1})^T = \begin{bmatrix} 1 & 0 & 0 & -1 \\ 0 & 2 & 0 & -2 \\ 0 & 0 & 2 & -2 \\ 0 & 0 & 0 & -1 \end{bmatrix}$$

6.11.3 Lambert's Cosine Law

The Light which strikes a surface point head-on (90^0 angle) is generally more intense than light that just glances a surface point; consider Figure 6.36 to understand this. Let a small shaft of incoming light with cross-sectional area given as dA. We can come up with a function which returns different intensities based on the alignment of the vertex normal and the light vector. When the vertex normal and light vector are perfectly aligned (i.e., the angle θ between them is 0°) the function should return maximum intensity and when the angle increases, the intensity diminishes accordingly. If the angle, θ > 90°, then the light strikes the back of a surface and so we set the intensity to zero. The Lambert's Cosine Law function is given as,

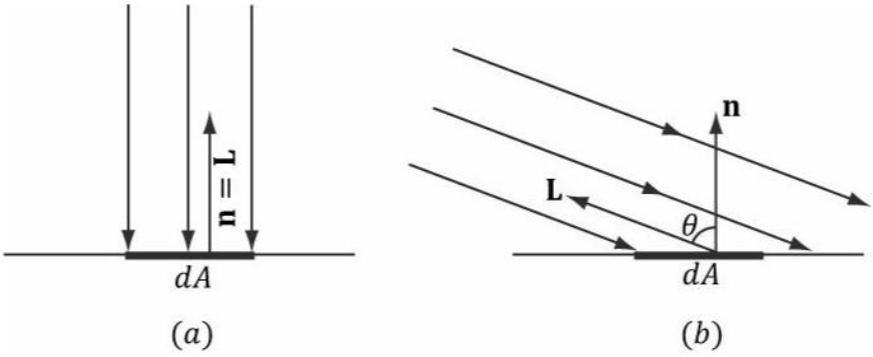


Figure 6.36. Consider a small area element dA. (a) The area dA receives the most light when the normal vector n and light vector L are aligned. (b) The area dA receives less light as the angle θ between n and L increases (as depicted by the light rays that miss the surface dA).

$f(\theta) = max(cos\theta,0) = max(L.n,0)$

where L and n are unit vectors. See Figure 6.9 shows a plot of f (θ) to see how the intensity, ranging from 0.0 to 1.0 (i.e., 0% to 100%), varies with θ.

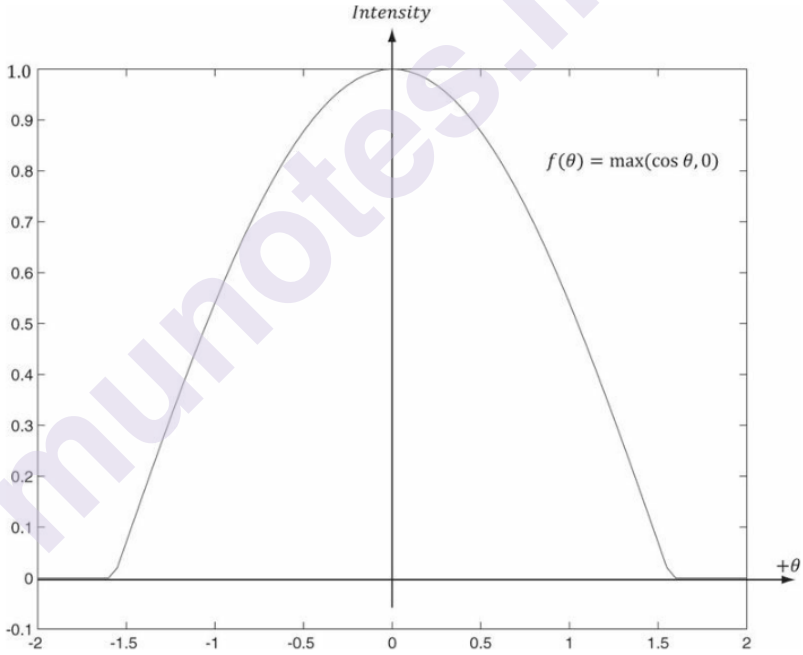


Figure 6.37. Plot of the function f (θ) = max (cos θ,0) = max (L · n, 0) for – 2 ≤ θ ≤ 2. Note that π/2 ≈ 1.57.

6.11.4 Diffuse Lighting

One of the common type of lighting is Diffuse Lighting. To understand this consider a rough surface, as shown in Figure 6.38. A diffuse reflection occurs when light strikes a point on such a rough surface, and light rays scatter in various random directions. In our modeling this kind of light/surface interaction, we stipulate that the light scatters equally in all directions above the surface. Similarly, the reflected light will reach the eye regardless of theeye position.

Hence, Diffuse lighting calculation is independent of view point, the color of the surface will always look the same no matter the viewpoint. We can do the diffuse light calculation in two parts: for the first, diffuse light color and a diffuse material color are specified. Amount of incoming diffuse light which the surface reflects and absorbs is specified by the diffuse material; and, this is handled with a component-wise color multiplication.

Consider for example, some point on a surface reflects 50% incoming red light, 100% green light, and 75% blue light, the incoming light color is 80% intensity white light. Then the incoming diffuse light color is given as: ι_d = (0.8, 0.8, 0.8) and the diffuse material color is given by md = (0.5, 1.0, 0.75); then the amount of light reflected off the point is given by:

$D = \iota_d \otimes m_d = (0.8,0.8,0.8) \otimes (0.5,1.0,0.75) = (0.4,0.8,0.6)$.

Finally, the Lambert's cosine law is included to finish this calculation.

Let ιd be the diffuse light color, md be the diffuse material color, and k_d = max (L · n, 0), where L is the light vector, and n is the surface normal. The amount of reflected diffuse light off the point is given by equation:

$c_d = k_d \cdot \iota_d \otimes m_d = k_d D$ (eq. 6.4)

Figure 6.38. Incoming light scatters in random directions when striking a diffuse surface. The idea is that the surface is rough at a microscopic level.

6.11.5 Ambient Lighting

Our lighting model will not take the light bounced off the other objects into consideration. But, in real world, most of the light that we will notice is the indirect kind. For example, consider 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.

For the calculation of this indirect light, we will use ambient term as given below in the lighting equation:

$\mathbf{A = \iota_a \otimes m_a}$

The color ι_a specifies total amount of indirect (ambient) light that a surface is receiving from the light source. The ambient material color denoted by ma gives the amount of incoming ambient light which the surface reflects and absorbs. Ambient light uniformly brightens up the object by a bit; hence, we cannot do a specific physical calculation. Here the indirect light will scatter and bounce in the scene many times and it strikes the object in every direction equally. If we combine both ambient and diffuse terms, we will get the new lighting equation:

$$\text{LitColor} = \iota_a \otimes m_a + k_d . \iota_d \otimes m_d$$

$$= A + k_d D \qquad \text{(eq 6.5)}$$

6.11.6 Specular Lighting

Consider a smooth surface, as shown in Figure 6.39. Light reflects sharply when it strikes such a surface in a direction through a cone of reflectance; this kind of reflection is known as specular reflection.

Specular light may not travel in our eyes as in the case of diffuse light; because specular light reflects in a specific direction. The specular lighting calculation is viewpoint dependent. It means if we move eye in the scene the amount of specular light it receives will change.

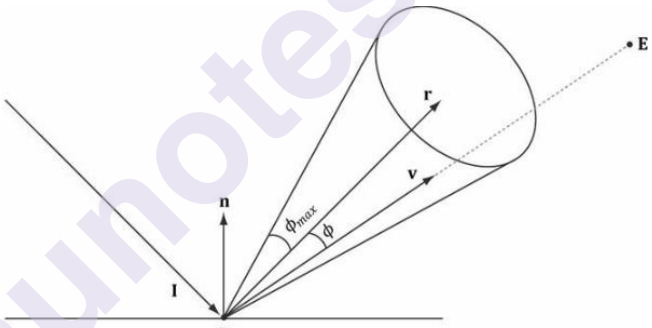


Figure 6.39. The incoming light ray is denoted by I. The specular reflection does not scatter in all directions, but instead reflects in a general cone of reflection whose size we can control with a parameter. If v is in the cone, the eye receives specular light; otherwise, it does not. The closer is aligned with the reflection vector r, the more specular light the eye receives.

The cone by which the specular light reflects through is given by an angle φmax with respect to the reflection vector r. It makes sense to vary the specular light intensity based on the angle φ between the reflected vector r and the view vector $= \frac{E-P}{|E-P|}$. We can deifne that the specular light intensity is maximized when φ = 0 and smoothly decreases to zero as φ approaches φmax. We can modify the Lambert's cosine law to represent this concept mathematically.

See Figure 6.40 to understand the graph of the cosine function for different powers of p ≥1. If we choose different p value, then we indirectly control the cone angle φmax where the light intensity drops to zero. The shininess of the surface can be controlled with the parameter p; means, highly

polished surfaces will have a smaller cone of reflectance than less shiny surfaces. Hence, we can use larger p value for shiny surface than for matte ones.

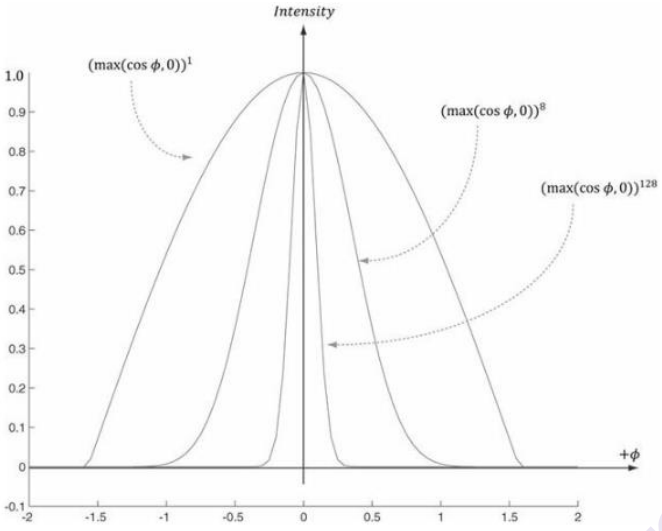


Figure 6.40. Plots of the cosine functions with different powers of p ≥ 1.

Take a note, that because v and r are the unit vectors, we have that cos (φ) = v · r. The amount of specular light reflected off a point that makes it into the eye is given by:

$C_s = k_s \,.\, \iota_s \otimes m_s$

$= k_s S$

Where

$$k_s = \begin{cases} \max(v.r, 0)^p, & L.n > 0 \\ 0, & L.n \leq 0 \end{cases}$$

The color ι_s gives the amount of specular light a light source is emitting. The specular material color m_s defines the specular light reflected by the surface. The factor k_s is used for scaling of the intensity of specular light dependent on angle between r and v. Consider Figure 6.41 which shows that, it is possible for a surface to receive no (zero) diffuse light (L · n < 0), but only receives specular light. However, if such is the case then it makes no sense for the surface to receive specular light, so we set $k_s = 0$ in this case.

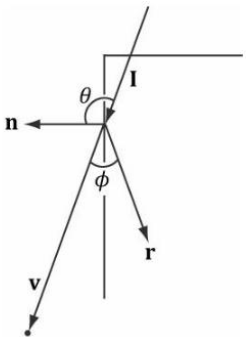


Figure 6.41. The eye can receive specular light even though the light strikes the back of a surface. This is incorrect, so we must detect this situation and set kS = 0 in this case.

Note that the specular power p should always be greater than or equal to 1.

Our new lighting model is:

$\text{LitColor} = \iota_a \otimes m_a + k_d . \iota_d \otimes m_d + k_s . \iota_s \otimes m_s$

$= A + k_d D + k_s S$

$k_d = \max(L.n,0)$

$$k_s = \begin{cases} \max(v.r,0)^p, & L.n > 0 \\ 0, & L.n \leq 0 \end{cases} \qquad \text{(eq. 6.6)}$$

Notes: The reflection vector is given by: r = I - 2 (n · I)n; see Figure 6.42.

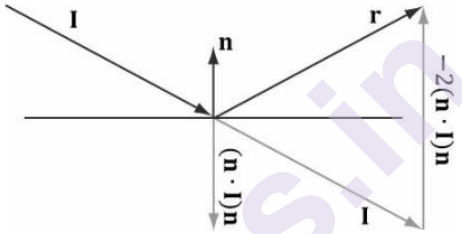


Figure 6.42. Geometry of reflection.

6.11.7 Specifying Materials

Material plays an important role in lighting. Depending on the surface, material values may vary; means, different points on the surface may have different material values, consider Figure 6.42 to understand this. As an example, consider a car model, where the frame, windows, lights, and tires reflect and absorb light differently, and so the material values would need to vary over the car surface.

To model this variation in material approximately, we can specify material values on the per vertex basis. Interpolation on these per vertex materials will be done across triangle during rasterization stage, which gives us material values for each point on the surface of the triangle mesh. Per vertex colors add additional data to our vertex structures, and we need to have tools to paint per vertex colors. We can set the material values to a member of a constant buffer, and all subsequently drawn geometry will use that material until it is changed between draw calls. The following pseudocode shows how we would draw the car:

Set Primary Lights material to constant buffer
Draw Primary Lights geometry
Set Secondary Lights material to constant buffer
Draw Secondary Lights geometry
Set Tire material to constant buffer
Draw Tire geometry
Set Window material to constant buffer

Draw Windows geometry
Set Car Body material to constant buffer
Draw car body geometry
Our material structure looks like this, and is defined in LightHelper.h:

```
struct Material
{
Material() { ZeroMemory(this, sizeof(this)); }
XMFLOAT4 Ambient;
XMFLOAT4 Diffuse;
XMFLOAT4 Specular; // w = SpecPower
XMFLOAT4 Reflect;

};
```

Reflect member will be used when mirror like reflections are used with mirror like surfaces. We embedded the specular power exponent p into the 4th component of the specular material color as the alpha component is not needed for lighting, so we might as well use the empty slot to store something useful. At the vertex level we specify normal to obtain a normal vector approximation at each point on the surface of the triangle mesh. These vertex normals will be interpolated across the triangle during rasterization. Let us now see parallel, point and spot lights in the following sections.

6.11.8 Parallel Lights

A parallel light (or directional light) approximates a light source that is very far away. We can also approximate all incoming light rays as parallel to each other.

A vector is used to define a parallel light source, which specifies the direction the light rays travel. The same direction vector is used by all light rays from the same source as light rays are parallel. The light vector aims in the opposite direction the light rays travel. A equation for a directional light is exactly as Equation 6.6.

6.11.9 Point Lights

A light bulb, which radiates spherically in all directions is an example of a point light. For an arbitrary point P, there exists a light ray originating from the point light position Q traveling toward the point. We define the light vector to go in the opposite direction; that is, the direction from the point P to the point light source Q:

$$L = \frac{Q - P}{| Q - P |}$$

The light vector calculation is the only differentiation factor in point lights and parallel lights; which is constant in parallel lights and varies for every point in point lights.

6.11.9.1 Attenuation

The light intensity weakens as a function of distance based on the inverse squared law. Also note that, the light intensity at a point a distance d away from the light source is given by:

$$I(d) = \frac{I_0}{d^2}$$

where I_0 is the light intensity at a distance $d = 1$ from the light source. This formula will not give perfect results always. Hence, instead of worrying about physical accuracy, we make a more general function that gives the artist/programmer some parameters to control.

To scale intensity we can use formula like:

$$I(d) = \frac{I_0}{a_0 + a_1 d + a_2 d^2}$$

We call a_0, a_1, and a_2 as the attenuation parameters, and they are to be supplied by the artist or programmer. If you actually want the light intensity to weaken with the inverse distance, then set $a_0 = 0$, $a_1 = 1$, and $a_2 = 0$. If you want the actual inverse square law, then set $a_0 = 0$, $a_1 = 0$, and $a_2 = 1$. If we add attenuation equation into the lighting equation we can have:

$$LitColor = A + \frac{I_d D + k_s S}{a_0 + a_1 d + a_2 d^2}$$

Interestingly, attenuation doesn't affect ambient term as the ambient term is used to model indirect light that has bounced around.

6.11.9.2 Range

In point lights, we include an additional range parameter. The point who has more distance from light source than the given range, will not receive any light from that source. It is useful for localizing a light to a particular area.

The attenuation parameter is useful to be able to explicitly define the max range of the light source. The range parameter is also useful in shader optimization. The range parameter does not affect parallel lights, which model light sources very far away.

6.11.10 Spotlights

Flash light is one of the good physical example of a spotlight. A spotlight has a position Q, is aimed in a direction d, and radiates light through a cone refer Figure 6.43.

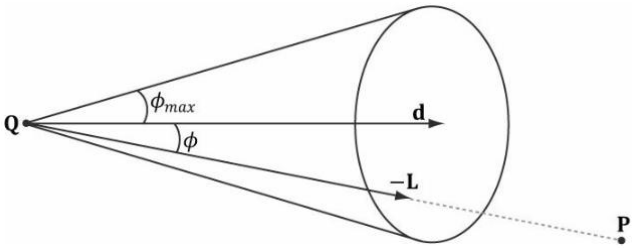


Figure 6.43. A spotlight has a position Q, is aimed in a direction d, and radiates light through a cone with angle φmax.

To implement a spotlight the light vector is given by:

$$L = \frac{Q - P}{| Q - P |}$$

where P is the position of the point being lit and Q is the position of the spotlight. Observe in Figure 6.43 that P is inside the spotlight's cone if and only if the angle φ between - L and d is smaller than the cone angle φmax.

All the light in the spotlight's cone should not be of equal intensity; the light at the center of the cone should be the most intense and the light intensity should fade to zero as φ increases from 0 to φmax. We use the following function which helps us to control the intensity falloff as a function of

φ: k_{spot} (φ) = max (cosφ,0)s = max (-L ·d, 0)s

You can see that, the intensity smoothly fades as φ increases; additionally, by altering the exponent s, we can indirectly control φmax (the angle the intensity drops to 0); that is to say we can shrink or expand the spotlight cone by varying s.

For example, if we set s = 8, the cone has approximately a 45° half angle. So the spotlight equation is just like the point light equation, except that we multiply by the spotlight factor to scale the light intensity based on where the point is with respect to the spotlight cone:

$$LitColor = k_{spot}\left(A + \frac{k_d D + k_s S}{a_0 + a_1 d + a_2 d^2}\right) \qquad \text{(eq.6.7)}$$

If we compare Equation 6.6 and 6.7, we can observe that, a spotlight is more expensive than a point light because we need to compute the k_{spot} factor and multiply by it. If we compare Equation 6.5 and 6.6, we can observe that, a point light is more expensive than a directional light because the distance *d* needs to be computed, and we need to divide by the attenuation expression.

To summarize, note that, directional lights are the least expensive light source, followed by point lights and spotlights are the most expensive light source.

6.11.11 Implementation

6.11.11.1 Lighting Structures

In LightHelper.h, we define the following structures to represent the three types of lights we support.

```
struct Directional Light
{
DirectionalLight() { ZeroMemory(this, sizeof(this)); }
XMFLOAT4 Ambient;
XMFLOAT4 Diffuse;
XMFLOAT4 Specular;
XMFLOAT3 Direction;
float Pad; // Pad the last float so we can
// array of lights if we wanted.
};
struct Point Light
{
PointLight() { ZeroMemory(this, sizeof(this)); }
XMFLOAT4 Ambient;
XMFLOAT4 Diffuse;
XMFLOAT4 Specular;
// Packed into 4D vector: (Position, Range)
XMFLOAT3 Position;
float Range;
// Packed into 4D vector: (A0, A1, A2, Pad)
XMFLOAT 3 Att;
float Pad; // Pad the last float so we can set an
// array of lights if we wanted.
};
struct SpotLight
{
SpotLight() { ZeroMemory(this, sizeof(this)); }
XMFLOAT4 Ambient;
XMFLOAT4 Diffuse;
XMFLOAT4 Specular;

// Packed into 4D vector: (Position, Range)
XMFLOAT3 Position;
float Range;
// Packed into 4D vector: (Direction,
Spot)XMFLOAT3 Direction;
float Spot;
// Packed into 4D vector: (Att, Pad)
XMFLOAT 3 Att;
float Pad; // Pad the last float so we can set an
// array of lights if we wanted.
};
```

1. Ambient: The amount of ambient light emitted by the light source.
2. Diffuse: The amount of diffuse light emitted by the light source.
3. Specular: The amount of specular light emitted by the light source.
4. Direction: The direction of the light.

5. Position: The position of the light.
6. Range: The range of the light. A point whose distance from the light source is greater than the range is not lit.
7. Attenuation: Stores the three attenuation constants in the format (a0, a1, a2) that control how light intensity falls off with distance.
8. Spot: The exponent used in the spotlight calculation to control the spotlight cone.

6.12 BLENDING

Consider Figure 6.44 to understand Blending concept.

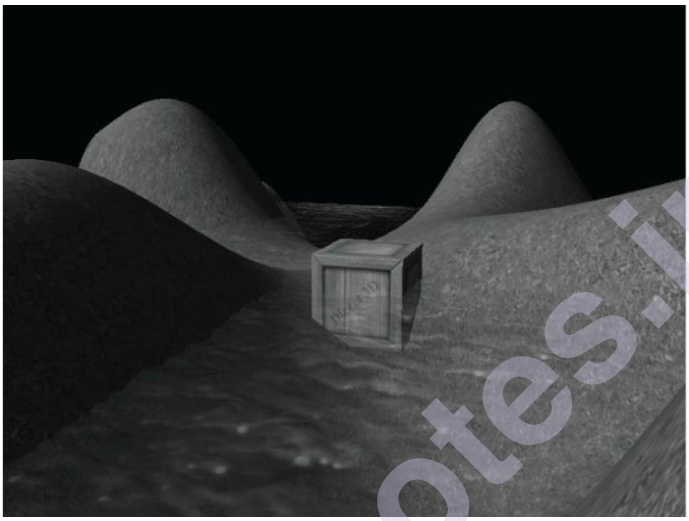


Figure 6.44. A semi-transparent water surface.

In order to render the scene given in the Figure, we start rendering the frame by first drawing the terrain (soil and surroundings) followed by the wooden crate, so that the terrain and crate pixels are on the back buffer. After that, draw the water surface to the back buffer using blending, hence the water pixels get blended (kind of mixed) with the terrain and crate pixels on the back buffer.

So we can see part of crate and terrain pixels through the water pixels as well. This is the power of using blending in the scene. We will now examine different blending techniques which allow us to blend (combine) the pixels that we are currently rasterizing (the source pixels) with the pixels that are already present on the back buffer (so-called destination pixels). This technique allows us to render semi-transparent objects such as water, glass, fog and gas.

6.12.1 The Blending Equation

Let us consider C_{src} as the color output from the pixel shader stage for the ij^{th} pixel which we are rasterizing (also called as source pixel), also let C_{dst} as the color of the ij^{th} pixel present on the back buffer (also called as destination pixel).

If we don't use blending, C_{src} would directly overwrite C_{dst} (by assuming that, it passes the depth/stencil test) and hence become the new color of the ij^{th} back buffer pixel. If we use blending, C_{src} and C_{dst} will be blended together to get the new color C that will overwrite the C_{dst}. Following blending equation will be used in Direct3D to blend source and destination pixels colors:

$C = C_{src} \otimes F_{src} \boxplus C_{dst} \otimes F_{dst}$

Here, the colors F_{src} (that is source blend factor) and F_{dst} (that is destination blend factor, and they also modify the original source and destination pixels in a variety of ways, hence achieving different effects. Here, the $\otimes$ operator is used to show component wise multiplication for the color vectors; and the $\boxplus$ operator may be any of the binary operators defined in next section. The blending equation we have seen before, holds only for the RGB components of the colors. The new alpha component is actually handled by a separate but similar equation:

$A = A_{src}F_{src} \boxplus A_{dst}F_{dst}$

This equation also is essentially the same, but it is possible in this that the blend factors and binary operation are different. To process RGB and alpha independently and differently, we need to separate them.

Note that blending the alpha components is needed much less frequently than blending the RGB components.

6.12.2 Blend Operations

The binary $\boxplus$ operator used in the blending equation may be one of the following:

```
typedef enum D3D11_BLEND_OP
{
D3D11_BLEND_OP_ADD = 1,
D3D11_BLEND_OP_SUBTRACT = 2,
D3D11_BLEND_OP_REV_SUBTRACT = 3,
D3D11_BLEND_OP_MIN = 4,
D3D11_BLEND_OP_MAX = 5,
} D3D11_BLEND_OP;
```

$C = C_{src} \otimes F_{src} + C_{dst} \otimes F_{dst}$

$C = C_{dst} \otimes F_{dst} - C_{src} \otimes F_{src}$

$C = C_{src} \otimes F_{src} - C_{dst} \otimes F_{dst}$

$C = \min(C_{src}, C_{dst})$

$C = \max(C_{src}, C_{dst})$

Note that the blend factors are ignored in the min/max operation.

For the alpha blending equation these same operators can be used. You can also specify a different operator for RGB and alpha. For example, it is possible to add the two RGB terms, but subtract the two alpha terms:

$C = C_{src} \otimes F_{src} + C_{dst} \otimes F_{dst}$

$A = A_{dst}F_{dst} - A_{src}F_{src}$

6.12.3 Blend Factors

Several blend operators are used for setting different combinations for the source and destination blend factors, and various different blending effects can be achieved. We will see some of the combinations later. The list given below gives the basic blend factors; these apply to both F_{src} and F_{dst}. You can refer to 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), $C_{dst} = (r_d,g_d,b_d)$, $A_{dst} = a_d$, (the RGBA values already stored in the render target), F being either F_{src} or F_{dst} and F being either F_{src} or F_{dst}, we have the following:

D3D11_BLEND_ZERO: F = (0,0,0) and F = 0D3D11_BLEND_ONE: F = (1,1,1) and F = 1D3D11_BLEND_SRC_COLOR: F = (rs, gs, bs)
D3D11_BLEND_INV_SRC_COLOR: F = (1 − rs, 1 − gs, 1 − bs)
D3D11_BLEND_SRC_ALPHA: F = (as, as, as) and F = as
D3D11_BLEND_INV_SRC_ALPHA: F = (1 − as, 1 − as, 1 − as) and F = 1 − as
D3D11_BLEND_DEST_ALPHA: F = (ad, ad, ad) and F = ad
D3D11_BLEND_INV_DEST_ALPHA: F = (1 − ad, 1 − ad, 1 − ad) and F = 1 − ad
D3D11_BLEND_DEST_COLOR: F = (rd, gd, bd)
D3D11_BLEND_INV_DEST_COLOR: F = (1 − rd, 1 − gd, 1 − bd)
D3D11_BLEND_SRC_ALPHA_SAT: F=(a'_s, a'_s, a'_s) and F = a'_s, and where $a'_s = clamp(a_s, 0, 1)$
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. 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. This allows you to specify the blend factor color to use directly; however, it is constant until you change the blend state.

6.12.4 Blend State

We have seen the blending operators and blend factors, but where can we set these values with Direct3D? The settings are controlled by the **ID3D11BlendState** interface. This interface can be 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**: This Pointer to the filled out D3D11_BLEND_DESC structure describing the blend state to create.

2. **ppBlendState**: This 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;

1. **AlphaToCoverageEnable:** Program will specify true to enable alpha-to-coverage, which is a multisampling technique useful when rendering foliage or gate textures. Program will specify false to disable alpha-to-coverage. This requires multisampling to be enabled.

2. **IndependentBlendEnable**: Total 8 render targets are supported by Direct3D 11 simultaneously. If this flag is set to true, then blending can be performed for each render target in a different way (different blend factors, different blend operations, blending disabled/enabled, etc.). When this flag is set to false, then all the render targets will be blended as described by the first element in the D3D11_BLEND_DESC::RenderTarget array. Multiple render targets are used for the advanced algorithms; for this instance, assume we only render to one render target at a time.

3. **RenderTarget**: The array of **D3D11_RENDER_TARGET_BLEND_DESC** elements (total 8), where the i^{th} element describes how blending is done for the i^{th} simultaneous render target. If **IndependentBlendEnable** is set to false, then all the render targets use **RenderTarget[0]** for blending. The **D3D11_RENDER_TARGET_BLEND_DESC** structure is defined like so:

typedef struct D3D11_RENDER_TARGET_BLEND_DESC {

BOOL BlendEnable; // Default: False
D3D11_BLEND SrcBlend; // Default: D3D11_BLEND_ONE
D3D11_BLEND DestBlend; // Default: D3D11_BLEND_ZERO

D3D11_BLEND_OP BlendOp; // Default: D3D11_BLEND_OP_ADD
D3D11_BLEND SrcBlendAlpha; // Default: D3D11_BLEND_ONE
D3D11_BLEND DestBlendAlpha; // Default: D3D11_BLEND_ZERO
D3D11_BLEND_OP BlendOpAlpha; // Default:
D3D11_BLEND_OP_ADD
UINT8 RenderTargetWriteMask; // Default:
D3D11_COLOR_WRITE_ENABLE_ALL

} D3D11_RENDER_TARGET_BLEND_DESC;
Here;

1. **BlendEnable**: Specifies true to enable blending and false to disable it.

2. **SrcBlend**: Is a member of the **D3D11_BLEND** enumerated type that specifies the source blend factor Fsrc for RGB blending.

3. **DestBlend**: Is a member of the **D3D11_BLEND** enumerated type that specifies the destination blend factor Fdst for RGB blending.

4. **BlendOp**: Is a member of the **D3D11_BLEND_OP** enumerated type that specifies the RGB blending operator.

5. **SrcBlendAlpha**: Is a member of the **D3D11_BLEND** enumerated type that specifies the destination blend factor Fsrc for alpha blending.

6. **DestBlendAlpha**: Is a member of the **D3D11_BLEND** enumerated type that specifies the destination blend factor Fdst for alpha blending.

7. **BlendOpAlpha**: Is a member of the **D3D11_BLEND_OP** enumerated type that specifies the alpha blending operator.

8. **RenderTargetWriteMask**: Is the combination of one or more of the following flags:
   typedef enum D3D11_COLOR_WRITE_ENABLE {

D3D11_COLOR_WRITE_ENABLE_RED = 1,
D3D11_COLOR_WRITE_ENABLE_GREEN = 2,
D3D11_COLOR_WRITE_ENABLE_BLUE = 4,
D3D11_COLOR_WRITE_ENABLE_ALPHA = 8,
D3D11_COLOR_WRITE_ENABLE_ALL =
(D3D11_COLOR_WRITE_ENABLE_RED |
D3D11_COLOR_WRITE_ENABLE_GREEN |
D3D11_COLOR_WRITE_ENABLE_BLUE |
D3D11_COLOR_WRITE_ENABLE_ALPHA)

} D3D11_COLOR_WRITE_ENABLE;

These flags are used for controlling which color channels in the back buffer are written to after blending. For example, we can disable writes to the RGB channels, and only write to the alpha channel, it is done by specifying **D3D11_COLOR_WRITE_ENABLE_ALPHA.** For advanced techniques this kind of flexibility is very useful. If blending is disabled, the color

returned from the pixel shader is used with no write mask applied. To bind a blend state object to the output merger stage of the pipeline, we call:

void ID3D11DeviceContext::OMSetBlendState(
ID3D11BlendState *pBlendState,
const FLOAT BlendFactor,

UINT SampleMask);

1. **pBlendState**: Is a pointer to the blend state object to enable with the device.

2. **BlendFactor**: Is an array of four floats defining an RGBA color vector. This color vector is used as a blend factor when

   D3D11_BLEND_BLEND_FACTOR or

   D3D11_BLEND_INV_BLEND_FACTOR is specified.

3. **SampleMask**: The 32 samples a multisampling can take is used with 32-bit integer value is used to enable/disable the samples. For example consider, if you turn off the 5th bit, then the 5th sample will not be taken. Of course, disabling the 5th sample only has any consequence if you are actually using multisampling with at least 5 samples. Generally the default of 0xffffffff is used, which does not disable any samples an application might take from being taken.

There is a default blend state (blending disabled); if we call **OMSetBlendState** with null, then it restores the default blend state. This blending requires additional per-pixel work, so only enable it if you need it, and turn it off when you are done.

The following code shows an example of creating and setting a blend state:

```
D3D11_BLEND_DESC transparentDesc = {0};
transparentDesc.AlphaToCoverageEnable = false;
transparentDesc.IndependentBlendEnable = false;
transparentDesc.RenderTarget[0].BlendEnable = true;
transparentDesc.RenderTarget[0].SrcBlend =
D3D11_BLEND_SRC_ALPHA;
transparentDesc.RenderTarget[0].DestBlend =
D3D11_BLEND_INV_SRC_ALPHA;
transparentDesc.RenderTarget[0].BlendOp =
D3D11_BLEND_OP_ADD;
transparentDesc.RenderTarget[0].SrcBlendAlpha =
D3D11_BLEND_ONE;
transparentDesc.RenderTarget[0].DestBlendAlpha =
D3D11_BLEND_ZERO;
transparentDesc.RenderTarget[0].BlendOpAlpha =
D3D11_BLEND_OP_ADD;
transparentDesc.RenderTarget[0].RenderTargetWriteMask =
D3D11_COLOR_WRITE_ENABLE_ALL;
```

```
ID3D11BlendState* TransparentBS;
HR(device->CreateBlendState(&transparentDesc,
&TransparentBS));
...
float blendFactors[] = {0.0f, 0.0f, 0.0f, 0.0f};
md3dImmediateContext->OMSetBlendState(
TransparentBS, blendFactor, 0xffffffff);
```

As with other state block interfaces, you should create them all at application initialization time, and then just switch between the state interfaces as needed. A blend state object can also be set and defined in an effect file:

```
BlendState blend

{

// Blending state for first render target.
BlendEnable[0] = TRUE;
SrcBlend[0] = SRC_COLOR;
DestBlend[0] = INV_SRC_ALPHA;
BlendOp[0] = ADD;
SrcBlendAlpha[0] = ZERO;
DestBlendAlpha[0] = ZERO;
BlendOpAlpha[0] = ADD;
RenderTargetWriteMask[0] = 0x0F;
// Blending state for second simultaneous render target.
BlendEnable[1] = True;
SrcBlend[1] = One;
DestBlend[1] = Zero;
BlendOp[1] = Add;
SrcBlendAlpha[1] = Zero;

DestBlendAlpha[1] = Zero;
BlendOpAlpha[1] = Add;
RenderTargetWriteMask[1] = 0x0F;

};
technique11 Tech
{

pass P0
{

...
// Use &amp;quot;blend&amp;quot; for this pass.
SetBlendState(blend, float4(0.0f, 0.0f, 0.0f, 0.0f), 0xffffffff);

}

}
```

The values you assign to the blend state object are like those you assign to the C++ structure, except without the prefix. For example, instead of specifying **D3D11_BLEND_SRC_COLOR** we just specify **SRC_COLOR** in the effect code. Understand also that the value assignments to the state properties are not case sensitive.

6.13 QUESTIONS:

1. Explain rendering pipeline.
2. What the input assembler stage?
3. Write note on vertex shader stage.
4. What is primitive topology?
5. What is frustum?
6. What Homogenous clip space?
7. What the tessellation stage?
8. Explain pixel shader.
9. Write a short note on meshes.
10. Write a note on texturing.
11. Write a note on blending.
12. Write a note lighting.

❄❄❄❄❄❄❄

7

INTERPOLATION AND CHARACTER ANIMATION

Unit Structure :

7.0 OBJECTIVES:

1. To understand and revise Trigonometry.
2. To know the concept of Interpolation.
3. Understanding the use of Interpolation.
4. Understanding Curves and their equation.
5. To know the use of Curves.
6. To understand Analytic geometry.
7. To understand intersection of circle with line.

7.1 TRIGONOMETRY

Trigonometry is one of the basic concept that we must understand when dealing with animation. If the word 'trigonometry' is split into its constituent parts, '*tri*' '*gon*' '*metry*', we see that it is to do with the measurement of three-sided polygons, means **triangles**.

Trigonometry is very old, and we need to understand for the analysis and solution of problems in computer graphics.

Functions provided by trigonometry are used in vectors, transforms, geometry, quaternions and interpolation.

Main purpose of trigonometry is to calculate the measurement of angles, it can be achieved by using two units of measurement: ***degrees*** and ***radians***.

The degree unit of measure derives from defining one complete rotation of 360˚. Every degree divides into 60 min, and every minute divides into 60 seconds.

Radian doesn't depend on any arbitrary constant. Radian is the angle created by a circular arc whose length is equal to the circle's radius. Because the perimeter of a circle is given by $2\pi r$, 2π radians correspond to one complete rotation. As we know, 360˚ correspond to 2π radians, 1 radian corresponds to $180^{\circ}/\pi$, which is approximately 57.3˚. Following is the relationship between radians and degrees:

$$\frac{\pi}{2} \equiv 90^0 \qquad \pi \equiv 180^0 \qquad \frac{3\pi}{2} \equiv 270^0 \qquad 2\pi \equiv 360^0$$

7.1.1 The Trigonometric Ratios:

Historically, the ancient civilizations knew that triangles, possessed some inherent properties, especially the ratios of sides and their associated angles.

If such ratios were known well in advance, the problems involving triangles with unknown lengths and angles could be computed by applying these ratios.

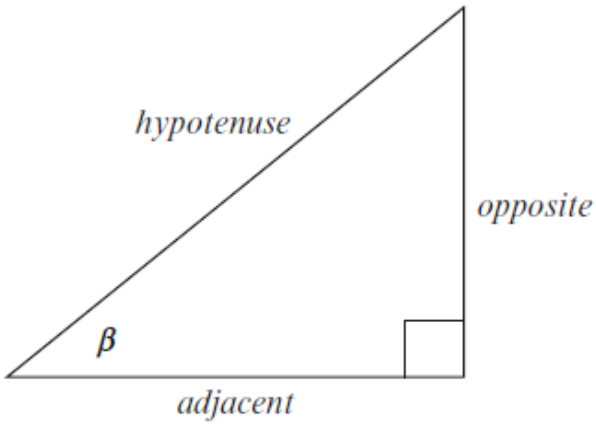


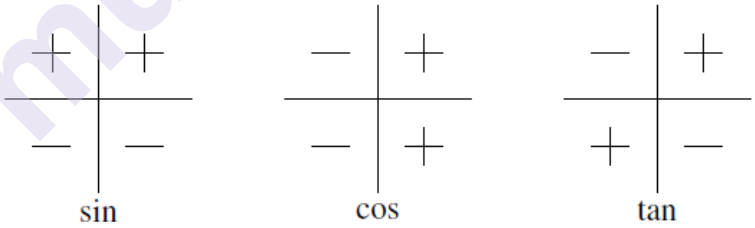
Fig. 7.1 labeling a right-angle triangle for the trigonometric ratios.

We all know the abbreviations sin, cos, tan, csc, sec, and cot are used in the trigonometric ratios. In Figure 7.1 a right-angled triangle is shown where the trigonometric ratios are given by:

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

$$csc\beta = \frac{1}{sin\beta} \qquad sec\beta = \frac{1}{cos\beta} \qquad cot\beta = \frac{1}{tan\beta}$$

The sin and cos functions have limits ±1, whereas tan has limits ±∞. The four quadrants are given with their signs as:



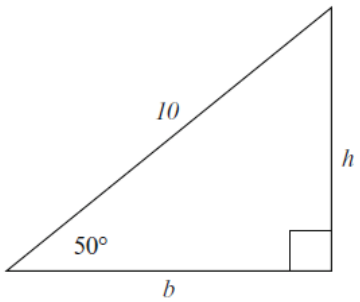


Fig. 7.2 *h* and *b* are unknown.

7.1.2 Example

Consider Figure 7.2 which shows another right-angled triangle where the hypotenuse and one angle are known. The calculation for the other side is done as follows:

$$\frac{h}{10} = \sin 50^0$$

$$h = 10 \sin 50^0 = 10\ X\ 0.76601 = 7.66$$

$$\frac{b}{10} = \cos 50^0$$

$$b = 10 \cos 50^0 = 10\ X\ 0.64279 = 6.4279$$

7.1.3 Inverse Trigonometric Ratios:

Every angle has its associated ratio in trigonometry, we need functions to convert one into the other.

The sin, cos and tan functions are used for conversion of angles into ratios, and respective the inverse functions **sin^{-1}**, **cos^{-1}** and **tan^{-1}** are used for conversion of ratios into angles.

Take example, $\sin 45^\circ \approx 0.707$, therefore $\sin^{-1} 0.707 \approx 45^\circ$.

As sine and cosine functions repeat indefinitely hence known as *cyclic* functions, their inverse functions return angles over a specific period.

7.1.4 Trigonometric Relationships:

We can find a strong relationship between the sin and cos definitions, and they are formally related by

$$\cos \beta = \sin(\beta + 90^0).$$

The theorem of Pythagoras also can be used to in some other formulas such as

$$\frac{\sin \beta}{\cos \beta} = \tan \beta$$

$$sin^2\beta + cos^2\beta = 1$$

$$1 + tan^2\beta = sec^2\beta$$

$$1 + cot^2\beta = csc^2\beta$$

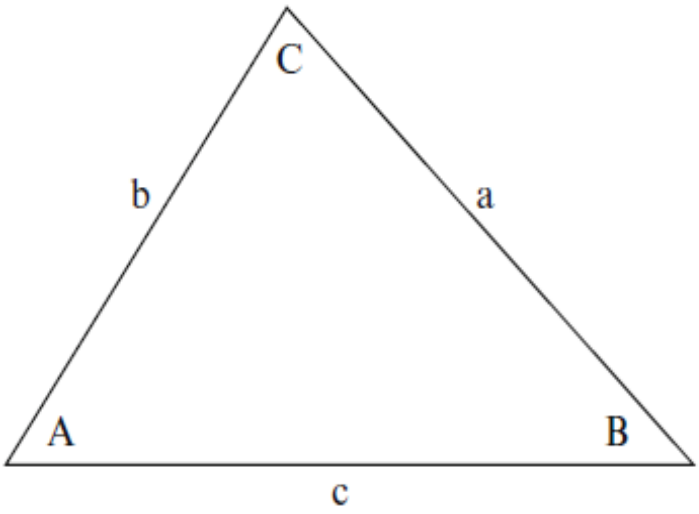


Fig. 7.3 An arbitrary triangle.

7.1.5 The Sine Rule:

Angles and the side lengths of a triangle can be related by using sine rule. In Figure 7.3 you can see a triangle labeled in such a way that side *a* is opposite angle *A*, side *b* is opposite angle *B*, etc.

We can form the sine rule for the Figure as

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

7.1.6 The Cosine Rule:

The cosine rule is used for expressing the $\sin^2\beta + \cos^2\beta = 1$ relationship for any arbitrary triangle as shown in Fig. 4.3. In three ways you can write cosine rule as follows:

$$a^2 = b^2 + c^2 - 2bc\cos A$$

$$b^2 = c^2 + a^2 - 2ca\cos B$$

$$c^2 = a^2 + b^2 - 2ab\cos C$$

And three more relationships also hold;

$$a = b\cos C + c\cos B$$

$$b = c\cos A + a\cos C$$

$$c = a\cos B + b\cos A$$

7.1.7 Compound Angles:

There are various sets of relationships which are compound trigonometric which shows how to add and subtract two different angles and multiplies of the same angle. Some of these most common relationships are given below:

$$\sin(A \pm B) = \sin A \cos B \ \pm \cos A \sin B$$

$$\cos(A \pm B) = \cos A \cos B \ \mp \ \cos A \sin B$$

$$\tan(A \ \pm B) = \frac{\tan A \ \pm \tan B}{1 \ \mp \tan A \tan B}$$

$$\sin 2\beta = 2 \sin \beta \cos \beta$$

$$\cos 2\beta = \ cos^2\beta - \ sin^2\beta$$

$$\cos 2\beta = 2 \ cos^2\beta - 1$$

$$\cos 2\beta = 1 - 2 \ sin^2\beta$$

$$\sin 3\beta = 3 \sin \beta - 4 \ sin^3 \beta$$

$$\cos 3\beta = 4 \ cos^3\beta - 3 \cos \beta$$

$$cos^2\beta = \frac{1}{2} (1 + \cos 2\beta)$$

$sin^2\beta = \frac{1}{2} (1 - \cos 2\beta)$.

7.1.8 Perimeter Relationships:

Refer Fig. 7.3, by using which we can create relationships those integrate angles with the perimeter of a triangle; are given as follows:

$$s = \frac{1}{2} (a + b + c)$$

$$\sin\left(\frac{A}{2}\right) = \sqrt{\frac{(s-b)(s-c)}{bc}}$$

$$\sin\left(\frac{B}{2}\right) = \sqrt{\frac{(s-c)(s-a)}{ca}}$$

$$\sin\left(\frac{C}{2}\right) = \sqrt{\frac{(s-a)(s-b)}{ab}}$$

$$\cos\left(\frac{A}{2}\right) = \sqrt{\frac{s(s-a)}{bc}}$$

$$\cos\left(\frac{B}{2}\right) = \sqrt{\frac{s(s-b)}{ca}}$$

$$\cos\left(\frac{C}{2}\right) = \sqrt{\frac{s(s-c)}{ab}}$$

$$\sin A = \frac{2}{bc}\sqrt{s(s-a)(s-b)(s-c)}$$

$$\sin B = \frac{2}{ca}\sqrt{s(s-a)(s-b)(s-c)}$$

$$\sin C = \frac{2}{ab}\sqrt{s(s-a)(s-b)(s-c)}$$

7.2 INTERPOLATION

Interpolation is a set of techniques in mathematics which is helpful to solve computer graphics problems. *Interpolation* is a technique to change one number to another.

Consider, for changing 2 to 4 we simply add 2 in the original value, which is very simple, hence not very useful. The interpolant function is usually used to change one number to another in 10 steps. For the previous example, we can start with 2 and repeatedly added 0.2, it would generate the sequence 2.0, 2.2, 2.4, 2.6, 2.8, 3.0, 3.2, 3.4, 3.6, 3.8, and eventually we reach to 4.

The interpolated numbers can be used in the applications where gradual or continuous change is needed like to scale, rotate, translate an object, moving the camera, changing color or brightness.

We can control this interval spacing in this interpolated values.

We will start the discussion with the simplest of all interpolants: the linear interpolant.

7.2.1 Linear Interpolation:

As we have seen, to create equal spacing between the interpolated values we will use *linear interpolant.* The example shown that increment 0.2 is calculated by subtracting the first number from the second and dividing the result by 10, i.e., (4−2)/10=0.2.

Let us express the same problem differently below.

If we take two numbers as $n1$ and $n2$, given as start and final values respectively, we will have the interpolated value which is controlled by a parameter t which ranges between 0 and 1. When the value of $t = 0$, then result is $n1$, and when the value of $t = 1$, then the result is $n2$. The solution can be given as:

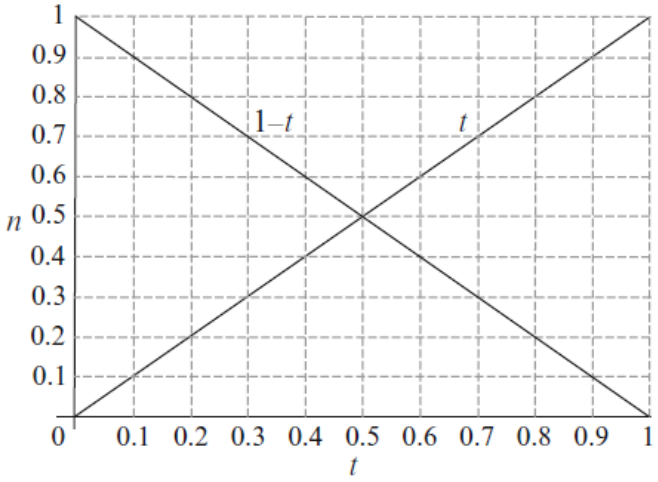


Fig. 7.4 The graphs of (1−t) and t over the range 0 to 1.

$$n = n_1 + t(n_2 - n_1)$$

For when $n_1 = 2$, $n_2 = 4$ and $t = 0.5$;

$$n = 2 + \frac{1}{2}(4 - 2) = 3$$

which is a halfway point. Moving forward, if $t = 0$, $n = n_1$, and if $t = 1$, $n = n_2$, these values confirm that we have a sound interpolant. We can express the equation differently as:

$$n = n_1(1 - t) + n_2 t \qquad (7.1)$$

Equation 7.1 shows what is really going on. See Figure 7.4 which shows the graphs of $(1-t)$ and t from the range 0 to 1. We can see that as t changes from 0 to 1, the graph of $(1-t)$ term also varies from 1 to 0. As a result the value of n_1 is attenuating to zero on the range of t when the term t scales the value of n_2 from zero to its actual value. The next Figure 7.5 shows these two actions with $n_1 = 1$ and $n_2 = 5$.

1. The terms $(1-t)$ and t sum to unity is one noticeable fact; and is not a coincidence. If this interpolant takes a quarter of n_1, it balances it with three quarters of n_2, and vice versa. We could obviously design an interpolant that takes arbitrary portions of n_1 and n_2, it also leads to some arbitrary results.

2. This simple interpolant is widely used in computer graphics software. For instance, consider the task of moving any object within the two locations (x_1, y_1, z_1) and (x_2, y_2, z_2). Then its interpolant position can be given as below:

$$x = x_1(1 - t) + x_2 t$$

$$y = y_1(1 - t) + y_2 t$$

$$z = z_1(1 - t) + z_2 t$$

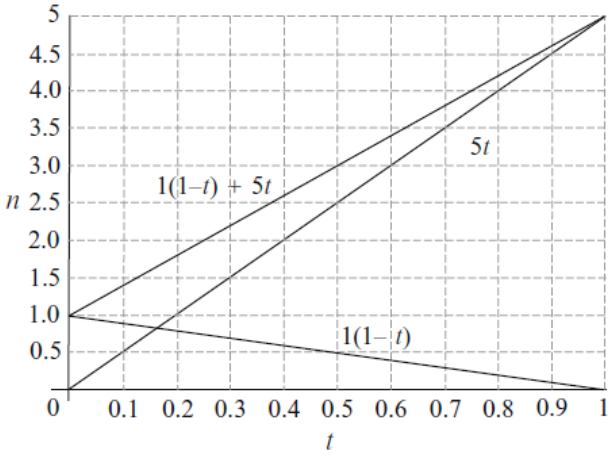


Fig. 7.5 The top line shows the result of linearly interpolating between 1 and 5.

Here, for $0 \leq t \leq 1$. Within the animation we can generate the parameter t from two frame values. By this kind of interpolant this can be assured that equal steps in t result in equal steps in x, y, and z. See Figure 7.6 which illustrates this linear spacing with a simple 2D example where we interpolate between the points (1,1) and (4,5). The spacing is equal between the intermediate interpolated points.

Fig. 7.6 Interpolating between the points (1,1) and (4,5).

Equation 7.1 can we given in matrix forma as follows:

$$n = [(1-t) \quad t].\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}$$

Or as

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

7.2.2 Non-Linear Interpolation:

Equal steps are ensured in the case of linear interpolant in the parameter *t*; but it is often required that equal steps in *t* may give rise to unequal steps in the interpolated values. This can be achieved by various mathematical techniques. For example, consider that we could use trigonometric functions or polynomials. Let's start with trigonometric solution.

7.2.2.1 Trigonometric Interpolation:

By trigonometry we know that, $\sin^2\beta + \cos^2\beta = 1$, this relation satisfies one of the requirements of an interpolant: the terms must sum to 1.

If the value of β varies between 0 to π/2, then $\cos^2\beta$ varies between 1 to 0, and value of $\sin^2\beta$ varies between 0 to 1, they can be used to modify the two interpolated values n_1 and n_2 as given below:

$$n = n_1 cos^2 t + n_2 sin^2 t \qquad [0 \leq t \leq \pi/2]. \tag{7.2}$$

The interpolation curves are shown in Fig 7.7 .

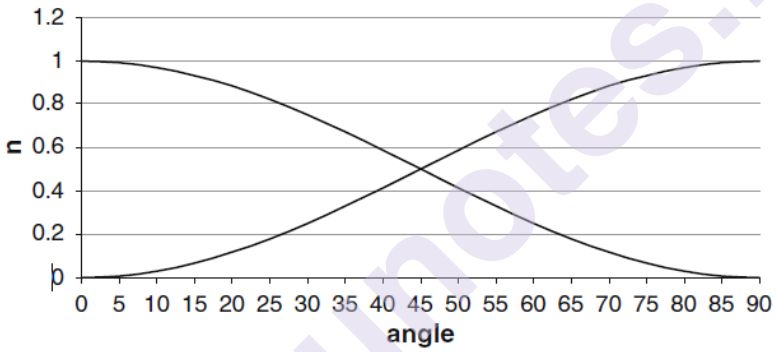


Fig. 7.7 The curves for $\cos^2\beta$ and $\sin^2\beta$.

Let's make the values of $n_1 = 1$ and $n_2 = 3$ which were placed in (7.2), the curves we can obtain are shown in Fig. 7.8.

If we use two 2D points in space like (1,1) and (4,3) and we apply this interpolant, we will obtain a straight-line interpolation, the distribution of points will be see as non-linear in Fig.7.9. Equal steps in *t* gives unequal distances.

The nature of curve is main problem of this approach here, because it is *sinusoidal,* and the slope is given by interpolated values. If we use a *polynomial,* we can gain control over this interpolated curve, which we will understand later.

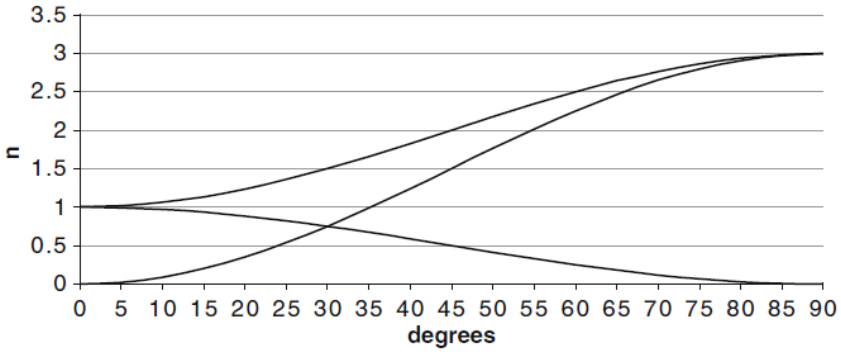


Fig. 7.8 Interpolating between 1 and 3 using a trigonometric interpolant.

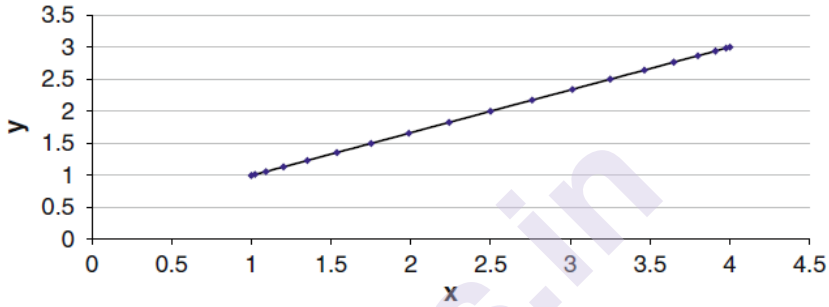


Fig. 7.9 Interpolating between two points (1,1) and (4,3). Note the non-linear distribution of points.

7.2.2.2 Cubic Interpolation:

In this type, we will develop a cubic blending function first which will be similar to the previous sinusoidal function. We can extend it to provide more flexibility. The basis for this interpolant is a *cubic polynomial*:

$$v_1 = at^3 + bt^2 + ct + d$$

The final interpolant can be given in the form of:

$$n = [v_1 \quad v_2].\begin{bmatrix} n_1 \\ n_2 \end{bmatrix}.$$

To find the values of the constants associated with the polynomials v_1 and v_2. The requirements are:

1. The growth of the cubic function v_2 must be from 0 to 1 for $0 \leq t \leq 1$.
2. The slope at a point t must equal the slope at the point $(1-t)$. The symmetry of slope is ensured by the range of function.
3. The value v_2 at any point t must also produce $(1-v_2)$ at $(1-t)$. Which ensures curve symmetry.

- The first requirement is satisfied by:

$$v_2 = at^3 + bt^2 + ct + d$$

when $t = 0$, $v_2 = 0$ and $d = 0$. Similarly, when $t = 1$, $v_2 = a+b+c$.

- The second requirement is satisfied when we differentiate v_2 to obtain the slope

$$\frac{dv_2}{dt} = 3at^2 + 2bt + c = 3a(1-t)^2 + 2b(1-t) + c$$

equating constants we discover $c = 0$ and $0 = 3a+2b$.

- The third requirement is satisfied by:

$$at^3 + bt^2 = 1 - [a(1-t)^3 + b(1-t)^2]$$

where we discover $1 = a+b$. But $0 = 3a+2b$, therefore $a = 2$ and $b = 3$.

Hence,

$$v_2 = -2t^3 + 3t^2 \qquad (7.3)$$

We can subtract equation (7.3) from 1 to find the curve's mirror curve, which starts at 1 and collapses to 0 as t moves from 0 to 1, as:

$$v_1 = -2t^3 - 3t^2 + 1.$$

Therefore, the two polynomials are

$$v_1 = -2t^3 - 3t^2 + 1. \qquad (7.4)$$

$$v_2 = -2t^3 + 3t^2 \qquad (7.5)$$

and are shown in Fig.7.9 . Those can be used as interpolants as:

$$n = v_1 n_1 + v_2 n_2$$

And the matrix form for the same is given as:

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

$$n = [t^3 \quad t^2 \quad t \quad 1] . \begin{bmatrix} 2 & -2 \\ -3 & 3 \\ 0 & 0 \\ 1 & 0 \end{bmatrix} . \begin{bmatrix} n_1 \\ n_2 \end{bmatrix} . \qquad (7.6)$$

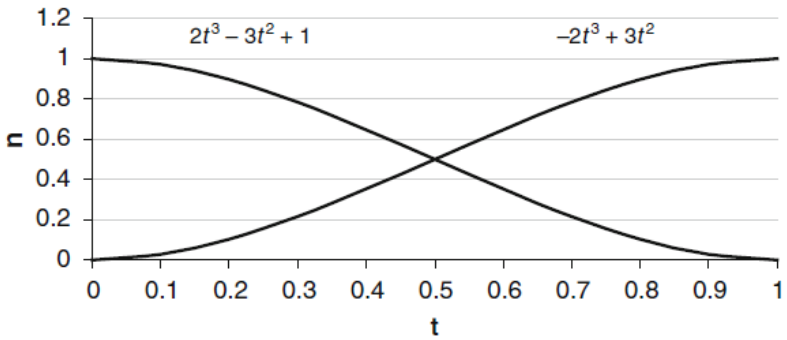


Fig. 7.10 Two cubic interpolants.

If we let $n_1 = 1$ and $n_2 = 3$ we obtain the curves shown in Fig. 7.11.

After applying the interpolant to the points (1,1) and (4,3) we obtain the curves which are shown in Fig. 8.9. Any pair of numbers can be blended together by using this interpolant.

Other qualities can be associated with the numbers n_1 and n_2, such as their tangent vectors s_1 and s_2. Perhaps we could interpolate these alongside n_1 and n_2. In fact this can be done, as we will see later.

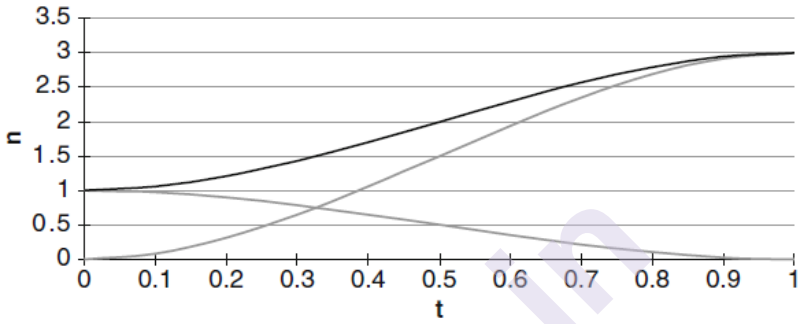


Fig. 7.11 Interpolating between 1 and 3 using a cubic interpolant.

The interpolating curve shown in Fig. 7.11 is to be modulated with two further cubic curves. One that blends out the tangent vector s_1 associated with n_1, and the other that blends in the tangent vector s_2 associated with n_2. We will start with a cubic polynomial to blend vector s_1 to zero:

$$v_{out} = at^3 + bt^2 + ct + d.$$

Here, v_{out} must equal zero when $t = 0$ and $t = 1$, otherwise it will disturb the start and end values. Therefore $d = 0$, and

$a+b+c = 0$.

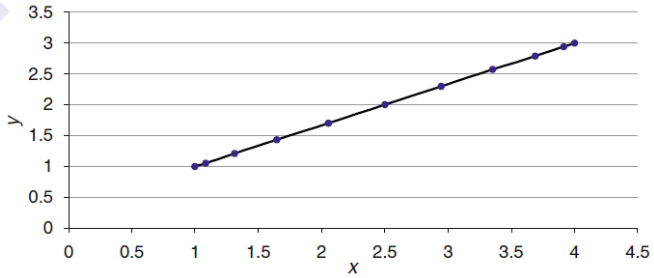


Fig. 7.12 A cubic interpolant between points (1,1) and (4,3).

The rate of change of v_{out} relative to t (i.e., dv_{out}/dt) must equal 1 when $t = 0$, so it can be used to multiply s1. If $t = 1$, then, dv_{out}/dt must equal 0 to attenuate any trace of s1:

$$\frac{dv_{out}}{dt} = 3at^2 + 2bt + c$$

But $\frac{dv_{out}}{dt} = 1$ when t=0 and $\frac{dv_{out}}{dt} = 0$ when t=1. Hence, c=1 and

3a+2b+1=0

Using equation 7.6 implies that *b*= -2 and *a*=1. Hence, the polynomial v_{out} has the form

$v_{out} = t^3 - 2t^2 + t$. *(7.7)*

using a similar argument, we can prove that the function to blen in s_2 equals

$v_{in} = t^3 - t^2$. *(7.8)*

Previous graphs are shown in Fig. 7.13. The complete interpolation function looks like

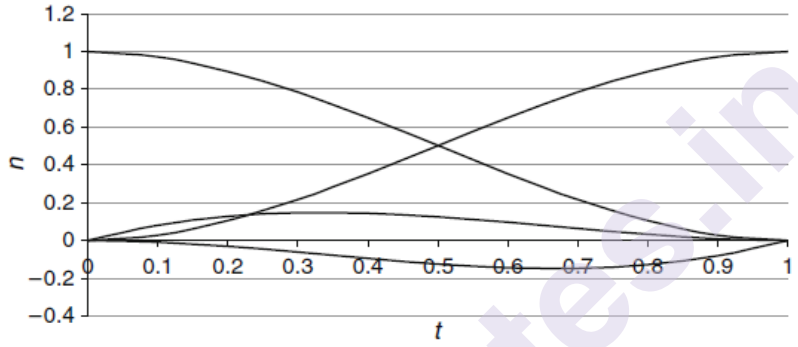


Fig. 7.13 The four Hermite interpolating curves.

$$n = [2t^3 - 3t^2 + 1 \quad -2t^3 + 3t^2 \quad t^3 - 2t^2 + t \quad t^3 - t^2].\begin{bmatrix} n_1 \\ n_2 \\ s_1 \\ s_2 \end{bmatrix}$$

And unpacking the constants and polynomial terms, we will get

$$n = [t^3 \quad t^2 \quad t^1 \quad 1].\begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix}.\begin{bmatrix} n_1 \\ n_1 \\ s_1 \\ s_2 \end{bmatrix}.$$

The interpolation given here is called ***Hermite interpolation*** type. The French mathematician Hermite also proved in 1873 that *e* is transcendental. To blend a pair of numerical values and their tangent vectors this interpolant can be used as shown above, or it can be used to interpolate between points in space. We will see one 2D example to demonstrate the latter part, it is easy in 3D too. The Figure 7.14 shows how the two points (0,0) and (1,1) are to be connected by a cubic curve which responds to the initial and final tangent vectors. At the start point (0,1) the tangent vector is $[-5\ 0]^T$, and at the final point (1,1) the tangent vector is $[0\ -5]^T$. The *x* and *y* interpolants are

$$x = [t^3 \quad t^2 \quad t^1 \quad 1]. \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 0 \\ 1 \\ -5 \\ 0 \end{bmatrix}.$$

$$y = [t^3 \quad t^2 \quad t^1 \quad 1]. \begin{bmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} 0 \\ 1 \\ 0 \\ -5 \end{bmatrix}.$$

Which becomes

$$x = [t^3 \quad t^2 \quad t^1 \quad 1]. \begin{bmatrix} -7 \\ 13 \\ -5 \\ 0 \end{bmatrix} = -7t^3 + 13t^2 - 5t$$

$$y = [t^3 \quad t^2 \quad t^1 \quad 1]. \begin{bmatrix} -7 \\ 8 \\ 0 \\ 0 \end{bmatrix} = -7t^3 + 8t^2.$$

When these polynomials are plotted over the range $0 \leq t \leq 1$ we obtain the curve shown in Fig.7.13.

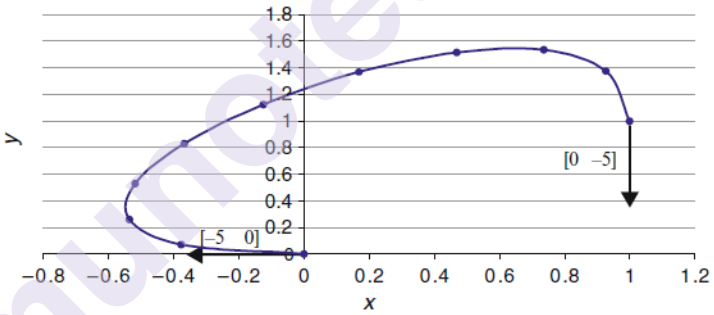


Fig. 7.14 A Hermite curve between the points (0,0) and (1,1) with tangent vectors

[−5 0]T and [0 −5]T .

We will now take a look at interpolating vectors.

7.2.3 Interpolating Vectors:

We have been interpolating between a pair of numbers in the previous sections. We can not use the same interpolants for vectors, because a vector contains both magnitude and direction, when we interpolating two vectors, both quantities must be preserved. Let us consider for example, if we interpolated the *x*- and *y*-components of the vectors $[2\ 3]^T$ and $[4\ 7]^T$, the in-between vectors would carry the change of orientation but ignore the change in magnitude. We must understand the required operation of interpolation to preserve both orientation and magnitudes in the result. From the Figure 7.15 we can see two unit vectors v_1 and v_2 which are separated

by an angle θ. A proportion of V_1 and a proportion of V_2 defines the interpolated vector v as given below:

$v = av_1+bv_2$.

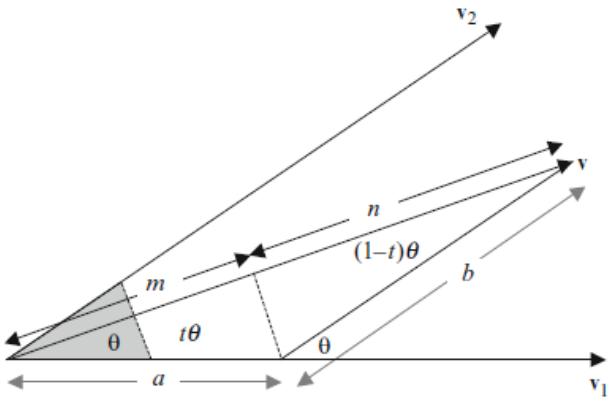


Fig. 7.15 Vector v is derived from a part of v1 and b part of v2.

Let's define the values of *a* and *b* such that they are a function of the separating angle θ. Vector v is *t*θ from v_1 and $(1-t)\theta$ from v_2, and we can understand from Fig.7.15 that using the sine rule;

$$\frac{a}{\sin(1-t)\theta} = \frac{b}{\sin t\theta} \qquad (7.9)$$

And further

$m = a\cos t\theta$

$n=b\cos(1 - t)\theta$

where

$m+n=1.$ *(7.10)*

from (7.9)

$$b = \frac{a \sin t\theta}{\sin(1-t)\theta}$$

From (7.10) we can have

$$a \cos t\theta + \frac{a \sin t\theta \cos(1-t)\theta}{\sin(1-t)\theta} = 1.$$

Solving for the value of *a* we can find

$$a = \frac{\sin(1-t)\theta}{\sin\theta}$$

$$b = \frac{\sin t\theta}{\sin\theta}.$$

Hence, the final interpolant is

$$v = \frac{\sin(1-t)\theta}{\sin\theta} v_1 + \frac{\sin t\theta}{\sin\theta} v_2. \qquad (7.11)$$

7.2.4 Interpolating Quaternions

The interpolants which are used for vectors, also works with quaternions. It means, if we have two quaternions as q_1 and q_2, the interpolated quaternion q is given by equation:

$$q = \frac{\sin(1-t)\theta}{\sin\theta} q_1 + \frac{\sin t\theta}{\sin\theta} q_2. \qquad (7.12)$$

This interpolant is applied individually to the four terms of the quaternion. θ is used as the angle between the two vectors which we are interpolating. It can also be derived using the dot product formula:

$$\cos\theta = \frac{v_1 . v_2}{|v_1||v_2|}$$

$$\cos\theta = \frac{x_1x_2 + y_1y_2 + z_1z_2}{|v_1||v_2|}$$

Also, when interpolating the quaternions, θ is computed by taking the 4D dot product of the two quaternions:

$$\cos\theta = \frac{q_1 . q_2}{|q_1||q_2|}$$

$$\cos\theta = \frac{s_1s_2 + x_1x_2 + y_1y_2 + z_1z_2}{|q_1||q_2|}$$

If we are using unit quaternions then,

$$\cos\theta = s_1s_2 + x_1x_2 + y_1y_2 + z_1z_2 \qquad (7.13)$$

We will now show how to interpolate with the pair of quaternions;

As an example, let us say we have two quaternions named as q_1 and q_2 that rotate 0° and 90° about the *z*-axis respectively, then:

q_1=[cos(0^0/2) + sin(0^0/2)[0i+0j+1k]]

q_2=[cos(90^0/2) + sin(90^0/2)[0i+0j+1k]]

which becomes

q_1 = [1+0i+0j+0k]

$q_2 \approx$ [0.7071+0i+0j+0.7071k] .

We can apply equation (7.12) to find any interpolated quaternion. Before that, we need to find the value of θ using equation (7.13) as:

cos θ ≈ 0.7071

$\theta = 45^0$

Now if $t = 0.5$, then interpolated quaternion is given by the equation,

$$q \approx \frac{\sin(45^0/2)}{\sin 45^0}[1 + 0i + 0j + 0k] + \frac{\sin(45^0/2)}{\sin 45^0}[1 + 0i + 0j + 0k]$$

$$\approx 0.541196[1 + 0i + 0j + 0k] + 0.541196[0.7071 + 0i + 0j + 0.7071k]$$

$$\approx [0.541196 + 0i + 0j + 0k] + [0.382683 + 0i + 0j + 0.382683k]$$

$$\approx [0.923879 + 0i + 0j + 0.382683k]$$

The interpolated quaternion is also unit quaternion, as the square root of sum of square is 1. It should rotate a point abut z-axis, halfway between 0^0 and 90^0, i.e., 45^0. Take a simple example to understand this:

Take point (1,0,0) and subject it to the standard quaternion operation:

$P' = qPq^{-1}$.

To keep the arithmatic work to a minimum, we substitute *a=0.923879* and *b=0.382683*.

Hence,

q= [*a +0i+0j+bk*]

q^{-1} =[*a -0i-0j-bk*]

P' = [*a +0i+0j+bk*]X[*0+1i+0j+0k*]X[*a-0i-0j-bk*]

=[*0+ai+bj+0k*]X[*a-0i-0j-bk*]

P' ≈ [0+0.7071i + 0.7071j + 0k]

Therefore, (1,0,0) is rotated to (0.7071,0.7071,0), which is correct!

7.3 CURVES

Here we investigate the foundations of curves. We can explore many of the ideas that are essential to understanding the mathematics behind 2D and 3D curves and how they are developed to produce surface patches.

7.3.1 The Circle

The circle equation can be given with the simple terms:

$x^2+y^2 = r^2$

where r is the radius and (x, y) is a point on the circumference. This equation is not very convenient for drawing the curve. For drawing curves we need two functions that generate the coordinates of any point on the circumference in terms of some parameter t. Figure 7.16 shows a scenario where the x- and y-coordinates are given by

$x = r\cos t$

$y = r\sin t \quad [\, 0 \le t \le 2\pi \,].$

If we vary the parameter t over the range 0 to 2π, we trace out the curve of the circumference. By selecting a suitable range of t we can isolate any portion of the circle's circumference.

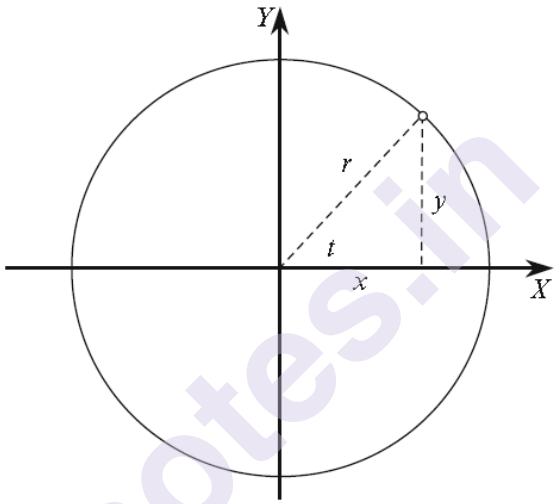


Fig. 7.16 The circle can be drawn by tracing out a series of points on the circumference.

7.3.2 The Ellipse

The equation for an ellipse is given as

$$\frac{x^2}{r_{maj}^2} + \frac{y^2}{r_{min}^2} = 1$$

and its parametric form is given as

$x = r_{maj} \cos t$

$y = r_{min} \sin t \qquad [\, 0 \le t \le 2\pi \,]$

where r_{maj} and r_{min} are the major and minor radii respectively, and (x, y) is a point on the circumference, as shown in Fig. 7.17.

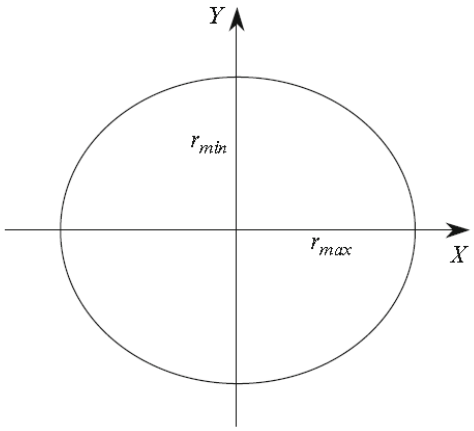


Fig. 7.17 An ellipse showing the major and minor radii.

7.4 BÉZIER CURVES

Bézier had become known for his special curves and surfaces.

7.4.1 Bernstein Polynomials

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

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-1} \qquad (7.14)$$

Where,

$$\binom{n}{i} = \frac{n!}{(n-i)!i!} \qquad (7.15)$$

where, if we put 3 for n!, 3! (factorial 3) is shorthand for 3 × 2 × 1. When (7.15) is evaluated for different values of *i* and *n*, we discover the pattern of numbers shown in Table 7.1. This pattern of numbers is known as the *Pascal's triangle*.

Table 7.1 Pascal's Triangle

	i						
n	0	1	2	3	4	5	6
0	1						
1	1	1					
2	1	2	1				
3	1	3	3	1			
4	1	4	6	4	1		
5	1	5	10	10	5	1	
6	1	6	15	20	15	6	1

The pattern represents the coefficients found in binomial expansions. For example, the expansion of $(x+a)^n$ for different values of n is

$(x+a)^0 = 1$

$(x+a)^1 = 1x+1a$

$(x+a)^2 = 1x^2+2ax+1a^2$

$(x+a)^3 = 1x^3+3ax^2+3a^2x+1a^3$

$(x+a)^4 = 1x^4+4ax^3+6a^2x^2+4a^3x+1a^4$

It produces Pascal's triangle as the polynomial coefficient terms.

Pascal, however, recognized other qualities in the numbers, in that they describe the odds governing combinations. For example, to determine the probability of any girl–boy combination in a family of six children, we sum the numbers in the 6th row of Pascal's triangle:

1+6+15+20+15+6+1 = 64.

The number (1) at the start and end of the 6th row represent the chances of getting six boys or six girls, i.e., 1 in 64. The next number (6) represents the next most likely combination: five boys and one girl, or five girls and one boy, i.e., 6 in 64. The center number (20) applies to three boys and three girls, for which the chances are 20 in 64.

The powers of t and $(1-t)$ in equation (7.14) appear as shown in Table 7.2 for different values of n and i. When the two sets of results are combined we get the complete Bernstein polynomial terms shown in Table 7.3.

Table 7.2 Expansion of the terms t and (1 - t)

	i				
n	0	1	2	3	4
1	t	$(1-t)$			
2	t^2	$t(1-t)$	$(1-t)^2$		
3	t^3	$t^2(1-t)$	$t(1-t)^2$	$(1-t)^3$	
4	t^4	$t^3(1-t)$	$t^2(1-t)^2$	$t(1-t)^3$	$(1-t)^4$

Table 7.3 The Bernstein polynomial terms

	i				
n	0	1	2	3	4
1	$1t$	$1(1-t)$			
2	$1t^2$	$2t(1-t)$	$1(1-t)^2$		
3	$1t^3$	$3t^2(1-t)$	$3t(1-t)^2$	$1(1-t)^3$	
4	$1t^4$	$4t^3(1-t)$	$6t^2(1-t)^2$	$4t(1-t)^3$	$1(1-t)^4$

As the sum of $(1-t)$ and t is 1,

$$[(1-t)+t]^n = 1 \qquad (7.16)$$

This is the reason we can use the binomial expansion of $(1-t)$ and t as interpolants. For example, when $n = 2$ we obtain the quadratic form

$$(1-t)^2+2t(1-t)+t^2 = 1. \qquad (7.17)$$

Figure 7.18 shows the graphs of the three polynomial terms of (7.17). The $(1-t)^2$ graph starts at 1 and decays to zero, whereas the t^2 graph starts at zero and rises to 1. The $2t(1-t)$ graph starts at zero reaches a maximum of 0.5 and returns to zero.

We can use these three terms to interpolate between a pair of values as follows

$$v = v1(1-t)^2+2t(1-t)+v^2t^2.$$

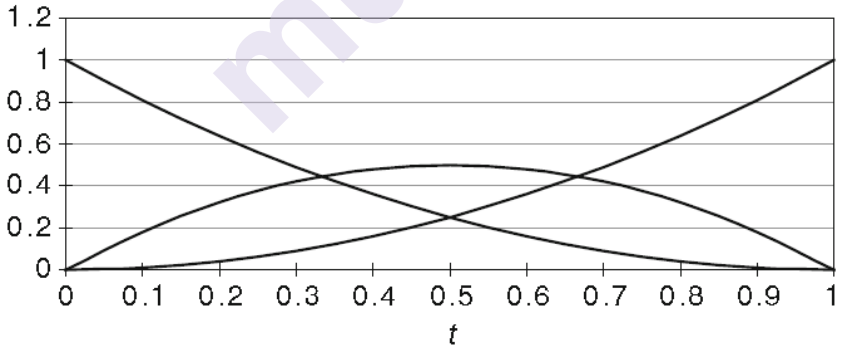


Fig. 7.18 The graphs of the quadratic Bernstein polynomials.

If $v1 = 1$ and $v2 = 3$ we obtain the curve shown in Fig.7.19. But there is nothing preventing us from multiplying the middle term $2t(1-t)$ by any arbitrary number vc:

$$v = v1(1-t)^2+vc^2t(1-t)+v^2t^2. \qquad (7.18)$$

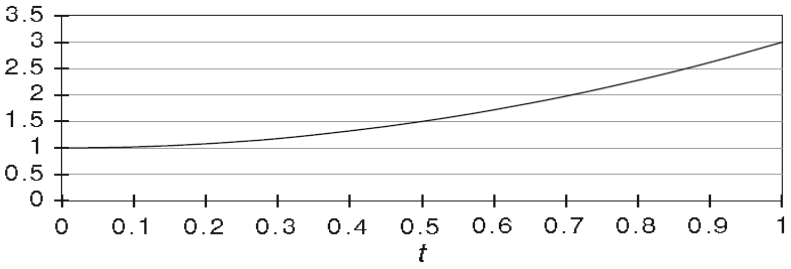


Fig. 7.19 Bernstein interpolation between the values 1 and 3.

For example, if $v_c = 3$ we obtain the graph shown in Fig.7.20 , which is totally different from Fig.7.19, with the value of v_c we can determining the shape of the curve between two values.

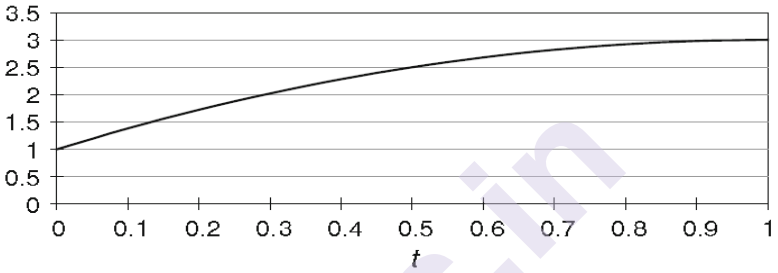


Fig. 7.20 Bernstein interpolation between the values 1 and 3 with $v_c = 3$.

Observe Figure 7.21 for a variety of graphs for different values of v_c. When the value of v_c is set midway between v_1 and v_2 very interesting results can be observed. Consider for example, when $v_1 = 1$, $v_2 = 3$ and $v_c = 2$, we obtain linear interpolation between v_1 and v_2, as shown in Fig.7.22.

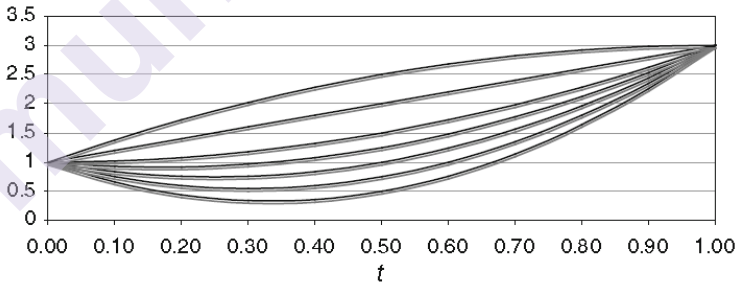


Fig. 7.21 Bernstein interpolation between the values 1 for different values of v_c.

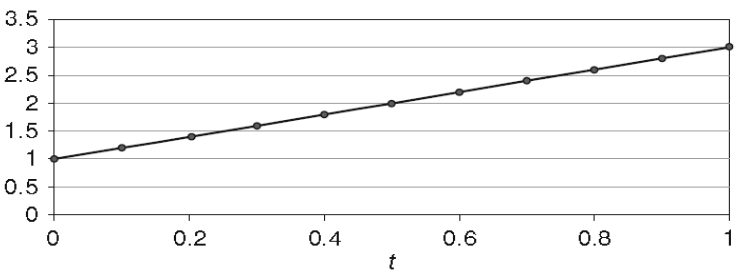


Fig. 7.22 Linear interpolation using a quadratic Bernstein interpolant.

7.4.2 Quadratic Bézier Curves

We can use Bernstein polynomials to form Quadratic Bézier curves which will be used to interpolate between the *x*-, *y*- and *z*-coordinates associated with the start- and end-points forming

the curve. Consider drawing a 2D quadratic Bézier curve between (1,1) and (4,3) using the equations as follows:

$$x = 1(1-t)^2+x_c^2t(1-t)+4t^2 \qquad (7.19)$$

$$y = 1(1-t)^2+y_c^2t(1-t)+3t^2. \qquad (7.20)$$

A Bézier curve has the interpolating and the approximating qualities: curve passes through the end points is the interpolating feature, while how the curve passes close to control point is determined by the approximating feature. Let's make x_c = 3 and y_c = 4 we obtain the curve shown as in Fig.7.23 , it shows how the curve intersects the end-points but miss the control point.

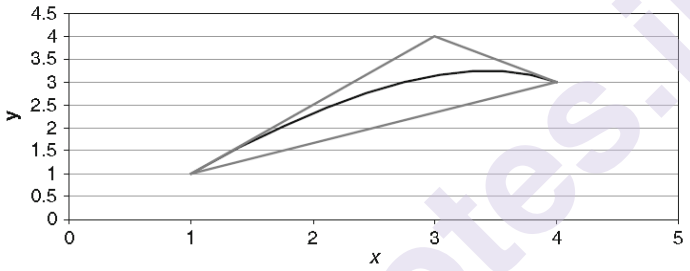


Fig. 7.23 Quadratic B´ezier curve between (1, 1) and (4,3), with (3, 4) as the control.

Bézier curves also has two important features of Bézier curves: the *convex hull* property, and the end slopes of the curve. The convex hull property states that the curve is always contained within the polygon connecting the start, end and control points. You can see in the diagram also that the curve is inside the triangle formed by the vertices (1,1), (3,4) and (4,3). Note also that the slope of the curve at (1,1) is equal to the slope of the line connecting the start point to the control point (3,4), and the slope of the curve at (4,3) is equal to the slope of the line connecting the control point (3,4) to the end point (4,3).

7.4.3 Cubic Bernstein Polynomials

We have to note two more important points:

1. No restrictions are placed upon the position of (x_c, y_c) – it can be anywhere.

2. Simply including *z*-coordinates for the start, end and control vertices creates 3D curves.

A *cubic curve* naturally supports one peak and one valley, which simplifies the construction of more complex curves.

When $n = 3$, we obtain the following terms:

$$[(1-t)+t]^3 = (1-t)^3+3t(1-t)^2+3t^2(1-t)+t^3$$

which can be used as a cubic interpolant, as

$$v = v_1(1-t)^3+v_{c1}3t(1-t)^2+v_{c2}3t^2(1-t)+v_2t^3.$$

See Figure 7.24 showing the graphs of the four polynomial terms.

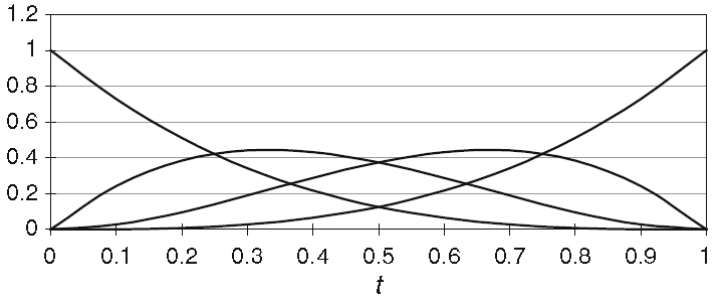


Fig. 7.24 The cubic Bernstein polynomial curves.

Consider two control values v_{c1} and v_{c2}. We can set any value, independent of the values chosen for v_1 and v_2. To illustrate this, let's consider an example of blending between values 1 and 3, with v_{c1} and v_{c2} set to 2.5 and −2.5 respectively. The blending curve is shown in Fig.7.25.

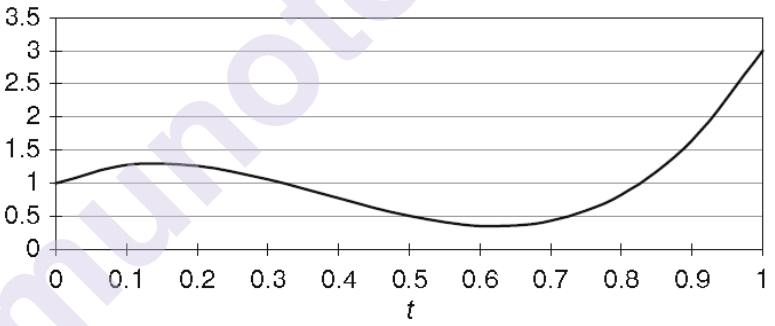


Fig. 7.25 The cubic Bernstein polynomial through the values 1, 2.5, –2.5, 3.

The next step is to associate the blending polynomials with *x*- and *y*-coordinates:

$$x = x_1(1-t)^3+x_{c1}3t(1-t)^2+x_{c2}3t^2(1-t)+x_2t^3 \qquad (7.21)$$

$$y = y_1(1-t)^3+y_{c1}3t(1-t)^2+y_{c2}3t^2(1-t)+y_2t^3. \qquad (7.22)$$

Evaluating (7.21) and (7.22) with the following points:

$(x_1, y_1) = (1,1)$ $(x_2, y_2) = (4,3)$

$(x_{c1}, y_{c1}) = (2,3)$ $(x_{c2}, y_{c2}) = (3,-2)$

See the guidelines between the end and control points that we obtain the cubic Bézier curve as shown in Fig.7.26.

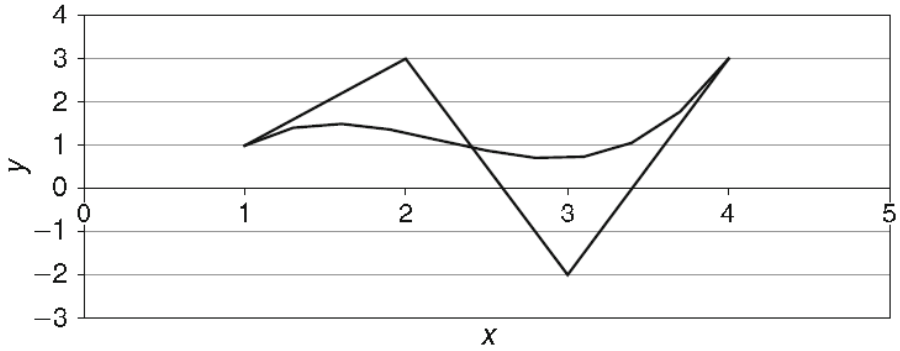


Fig. 7.26 A cubic Bézier curve.

let's set the values to

$(x_1, y_1) = (1,1)$ $(x_2, y_2) = (4,3)$

$(x_{c1}, y_{c1}) = (2,1.666)$ $(x_{c2}, y_{c2}) = (3,2.333)$

where (x_{c1}, y_{c1}) and (x_{c2}, y_{c2}) are points one-third and two-thirds respectively, between the start and final values. The single control point was halfway between the start and end values, we obtain linear interpolation as shown in Fig.7.27 .

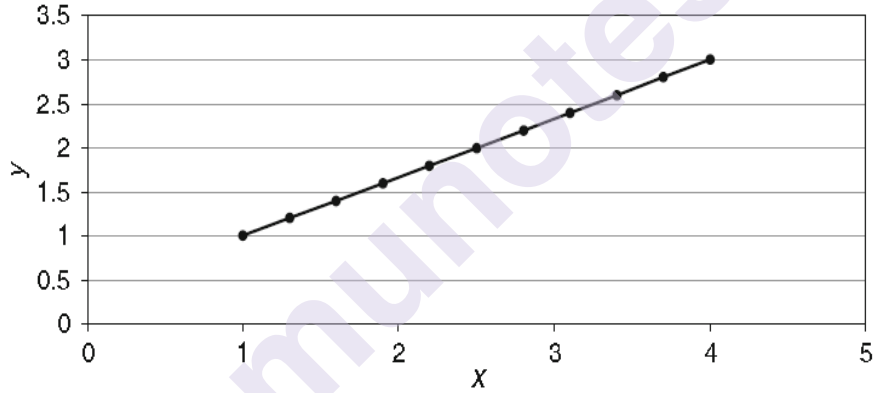


Fig. 7.27 A cubic Bézier line.

Equations (7.19) and (7.20) describe the three polynomial terms for generating a quadratic B´ezier curve and (7.21) and (7.22) describe the four polynomial terms for generating a cubic B´ezier curve. Quadratic equations are called *second-degree equations*, and cubics are called *third-degree equations*. In the original Bernstein formulation,

$$B_i^n(t) = \binom{n}{i} t^i (1-t)^{n-1} \qquad (7.23)$$

Here n represents the degree of the polynomial, and i, which has values between 0 and n, creates the individual polynomial terms.

If these points are stored as a vector P, the position vector p(t) for a point on the curve can be written as

$$p(t) = \binom{n}{i} t^i (1-t)^{n-i} P_i \qquad for\ [0 \le i \le n]$$

Or

$$p(t) = \sum_{i=0}^{n} \binom{n}{i} t^i (1-t)^{n-i} P_i \qquad for\ [0 \le i \le n] \qquad (7.24)$$

Or

$$p(t) = \sum_{i=0}^{n} B_i^n(t) P_i \qquad for\ [0 \le i \le n]. \qquad (7.25)$$

Let, a point **p**(*t*) on a quadratic curve is represented as,

$$p(t) = 1t^0(1-t)^2 P_0 + 2t^1(1-t)^1 P_1 + 1t^2(1-t)^0 P_2.$$

You will discover (7.24) and (7.25) used in more advanced books to describe Bézier curves.

7.4.4 A Recursive Bézier Formula

Note that the equation (7.24) describes the polynomial terms needed to create the blending terms. With the use of *recursive functions*, it is possible to arrive at another formulation that leads towards an understanding of *B-splines*.

As the coefficients of any row in Pascal's triangle are the sum of the two coefficients immediately above, we can write

$$\binom{n}{i} = \binom{n-1}{i} + \binom{n-1}{i-1}.$$

Hence, we can write

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

$$B_i^n(t) = (1-t) B_i^{n-1}(t) + t B_{i-1}^{n-1}(t).$$

When the degree is zero this process terminate; as all the recursive functions will terminate somewhere.

7.4.5 Bézier Curves Using Matrices

Matrices provide a very compact notation for algebraic formulae. Recall (7.17) which defines the three terms associated with a quadratic Bernstein polynomial. These can be expanded to

$(1-2t+t^2)\ (2t-2t^2)\ (t^2)$

and can be written as the product:

$$[t^2 \quad t \quad 1].\begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix}.$$

This means that equation (7.18) can be expressed as

$$v = [t^2 \quad t \quad 1] = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix} . \begin{bmatrix} v_1 \\ v_c \\ v_2 \end{bmatrix}$$

Or

$$p(t) = [t^2 \quad t \quad 1] = \begin{bmatrix} 1 & -2 & 1 \\ -2 & 2 & 0 \\ 1 & 0 & 0 \end{bmatrix} . \begin{bmatrix} P_1 \\ P_c \\ P_2 \end{bmatrix}$$

where p(t) points to any point on the curve, and P_1, P_c and P_2 point to the start, control and end points respectively.

A similar development can be used for a cubic Bézier curve, and given by following matrix:

$$p(t) = [t^3 \quad t^2 \quad t \quad 1] = \begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{bmatrix} . \begin{bmatrix} P_1 \\ P_{c1} \\ P_{c2} \\ P_2 \end{bmatrix}$$

7.4.5.1 Linear Interpolation

To interpolate linearly between two values $v0$ and $v1$ we use the following interpolant:

$v(t) = v_0(1-t)+v_1t$ for $[0 \leq t \leq 1]$.

Now let's invent a linear blending function and we want to compute the influence of the three values on any interpolated value $v(t)$ as follows:

$$v(t) = B_0^1(t)v_0 + B_1^1(t)v_1 + B_2^1(t)v_2. \qquad (7.26)$$

You can note that v_0 will influence $v(t)$ only when t is between t_0 and t_2. Also, v_1 and v_2 will influence $v(t)$ only when t is between t_1 and t_3, and t_2 and t_4 respectively.

When $t_1 \leq t \leq t_3$, the function must return a value reflecting the proportion of v_1 that influences $v(t)$. During the span $t_1 \leq t \leq t_2$, v_1 has to be blended in, and during the span $t_1 \leq t \leq t_3$, v_1 has to be blended out. The blending in is effected by the ratio

$$\left(\frac{t - t_1}{t_2 - t_1}\right)$$

and the blending out is effected by the ratio

$$\left(\frac{t_3 - t}{t_3 - t_2}\right).$$

Let's remind ourselves of this requirement by subscripting the ratios accordingly:

$$B_1^1(t) = \left(\frac{t - t_1}{t_2 - t_1}\right)_{1,2} + \left(\frac{t_3 - t}{t_3 - t_2}\right)_{2,3}.$$

We can now write the other two blending terms $B^1{}_0\,(t)$ and $B^1{}_2(t)$ as the final equations:

$$B_0^1(t) = \left(\frac{t - t_0}{t_1 - t_0}\right)_{0,1} + \left(\frac{t_2 - t}{t_2 - t_1}\right)_{1,2}$$

$$B_2^1(t) = \left(\frac{t - t_2}{t_3 - t_2}\right)_{2,3} + \left(\frac{t_4 - t}{t_4 - t_3}\right)_{3,4}.$$

7.5 B-SPLINES

B-splines also use polynomials to generate a curve segment and employ a series of control points that determine the curve's local geometry.

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 given as:

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

It may lead to some problems, but the division by a second polynomial brings certain advantages as listed below:

- They are used for describing perfect circles, ellipses, parabolas and hyperbolas, whereas nonrational curves can only approximate these curves.
- The polynomials are invariant of their control points when subjected to rotation, scaling, translation and perspective transformations, but in case of non-rational curves, they lose this geometric integrity.
- They also allow weights to be used at the control points to push and pull the curve.

Let's begin with uniform B-splines.

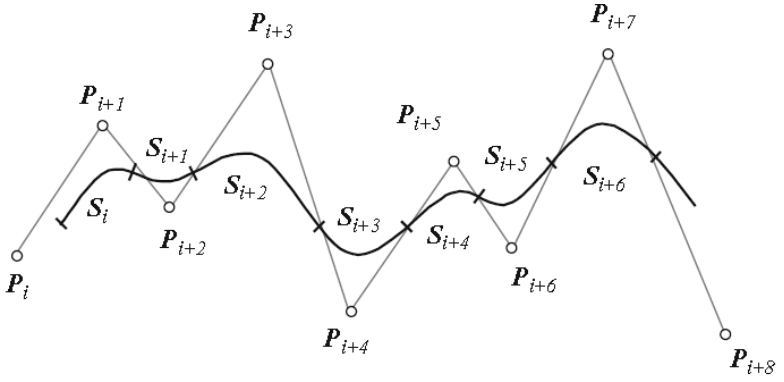


Fig. 7.28 The construction of a uniform non-rational B-spline curve.

7.5.1 Uniform B-Splines

A B-spline can be created from a string of curve segments in which the geometry is determined by a group of local control points. These several curves are known as *piecewise polynomials*. This curve segment doesn't pass through a control point.

Cubic B-splines provide a geometry that is one step away from simple quadratics, and possess continuity characteristics that make the joins between the segments invisible. To understand the construction consider the scenario as given in Fig.7.28 . In the diagram observe a group of (m+1) control points P_0, P_1, P_2, ..., P_m which determine the shape of a cubic curve constructed from a series of various curve segments S_0, S_1, S_2, ..., S_{m-3} .

The cubic curve segment S_i is influenced by P_i, P_{i+1}, P_{i+2}, P_{i+3}, and curve segment S_{i+1} is influenced by P_{i+1}, P_{i+2}, P_{i+3}, P_{i+4}. For the (m+1) control points, there are ($m-2$) curve segments.

A single segment $S_i(t)$ of a B-spline curve can be given by an equation:

$$S_i(t) = \sum_{r=0}^{3} P_{i+r} B_r(t) \quad for \quad [0 \le t \le 1]$$

Where

$$B_0(t) = \frac{-t^3+3t^2-3t+1}{6} = \frac{(1-t)^3}{6} \qquad (7.27)$$

$$B_1(t) = \frac{3t^3-6t^2+4}{6} \qquad (7.28)$$

$$B_2(t) = \frac{-3t^3+3t^2+3t+1}{6} \qquad (7.29)$$

$$B_3(t) = \frac{t^3}{6}. \qquad (7.30)$$

These B-spline *basis functions* and are shown in Fig. 7.29.

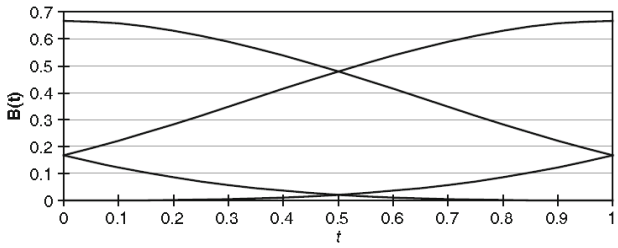


Fig. 7.29 The B-spline basis functions.

These four curve segments are part of one curve. The basis function $B_3(t)$ starts at zero and rises to 0.1666 at t = 1. It is taken over by $B_2(t)$ at t = 0, which rises to 0.666 at t = 1. The next segment is $B_1(t)$ and takes over at t = 0 and falls to 0.1666 at t = 1. Finally, $B_0(t)$ takes over at 0.1666 and falls to zero at t = 1. The above equations can be represented in matrix form as:

$$Q_1(t) = [t^3 \quad t^2 \quad t \quad 1]\frac{1}{6}\begin{bmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{bmatrix}.\begin{bmatrix} P_i \\ P_{i+1} \\ P_{i+2} \\ P_{i+3} \end{bmatrix}. \quad (7.31)$$

Let's understand how (7.31) works. We see the control points P_i, P_{i+1}, P_{i+2}, etc. Let's consider these be (0,1), (1,3), (2,0), (4,1), (4,3), (2,2) and (2,3). We can see them in Fig.7.30 connected together by straight lines. Consider first four control points: (0,1), (1,3), (2,0), (4,1), and subject the *x*- and *y*-coordinates to the matrix in (7.31) over the range $0 \leq t \leq 1$ we obtain the first B-spline curve segment shown in Fig.7.30 . If we move along one control point and take the next group of control points (1,3), (2,0), (4,1), (4,3), we obtain the second B-spline curve segment. This is repeated a further two times.

Observe Figure 7.30 which shows the four curve segments using two gray scales, and it is obvious that even though there are four discrete segments, which are joined perfectly.

Fig. 7.30 Four curve segments forming a B-spline curve.

7.5.2 Continuity

If the slope of the abutting curves match then then constructing curves from several segments can only succeed. It will be 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. Let's explore such features.

The *first level* of curve continuity C^0, ensures that the physical end of one basis curve corresponds with the following, e.g., $S_i(1) = S_{i+1}(0)$. We know that this occurs from the basis graphs as you can see in Fig.7.29. The *second level* of curve continuity C^1, ensures that the slope at the end of one basis curve matches that of the following curve. Basis functions can be used to confirm this:

$$B'_0(t) = \frac{-3t^2+6t-3}{6} \quad (7.31)$$

$$B'_1(t) = \frac{9t^2-12t}{6} \quad (7.32)$$

$$B'_2(t) = \frac{-9t^2+6t+3}{6} \qquad (7.33)$$

$$B'_3(t) = \frac{3t^2}{6}. \qquad (7.34)$$

If we evaluate (7.31) – (7.34) for the values $t = 0$ and $t = 1$, we will get the slopes 0.5, 0, −0.5, 0 for the joins between B_3, B_2, B_1, B_0. Then *third level* of curve continuity C^2, ensures that the rate of change of slope at the end of one basis curve matches that of the following curve. It can be confirmed by further differentiation:

$$B''_0(t) = -t + 1 \qquad (7.35)$$

$$B''_1(t) = 3t - 2 \qquad (7.36)$$

$$B''_2(t) = -3t + 1 \qquad (7.37)$$

$$B''_3(t) = t\,. \qquad (7.38)$$

After evaluating equations (7.35)–(7.38) for $t = 0$ and $t = 1$, we can get the values 1, 2, 1, 0 for the joins between B_3, B_2, B_1, B_0. These combined continuity results are tabulated in Table 7.4.

Table 7.4 Continuity properties of cubic B-splines

	t			t			t	
C^0	0	1	C^1	0	1	C^2	0	1
$B_3(t)$	0	1/6	$B'_3(t)$	0	0.5	$B''_3(t)$	0	1
$B_2(t)$	1/6	2/3	$B'_2(t)$	0.5	0	$B''_2(t)$	1	−2
$B_1(t)$	2/3	1/6	$B'_1(t)$	0	−0.5	$B''_1(t)$	−2	1
$B_0(t)$	1/6	0	$B'_0(t)$	−0.5	0	$B''_0(t)$	1	0

7.5.3 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.

7.5.4 Non-uniform Rational B-Splines

Non-uniform rational B-splines (NURBS) combine the advantages of non-uniform 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 modeling of computer animation characters.

7.6 ANALYTIC GEOMETRY

In computer graphics, basic elements of geometry and analytic geometry are frequently used. We will see some important concepts of analytic geometry in this section.

7.6.1 Review of Geometry

Here we will see the Euclidian's geometry. Although none of these developments affect computer graphics, they do place Euclid's theorems in a specific context: a set of axioms that apply to flat surfaces. We have probably all been taught that parallel lines don't meet, and that the internal angles of a triangle sum to 180◦, but these are only true in specific situations. As soon as the surface or space becomes curved, such rules break down.

7.6.1.1 Angles

As we know, 360◦ or 2π [radians] measure one revolution. We also must know how to convert from one to other.

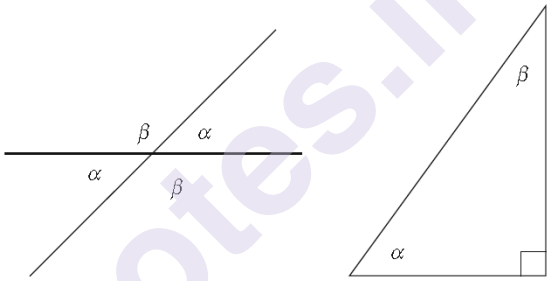


Fig. 7.31 Examples of adjacent, supplementary, opposite and complementary angles.

Observe Figure 7.31 which shows the examples of *adjacent / supplementary* angles (which sum to 180◦), *opposite* angles (equal), and *complementary* angles (sum to 90◦).

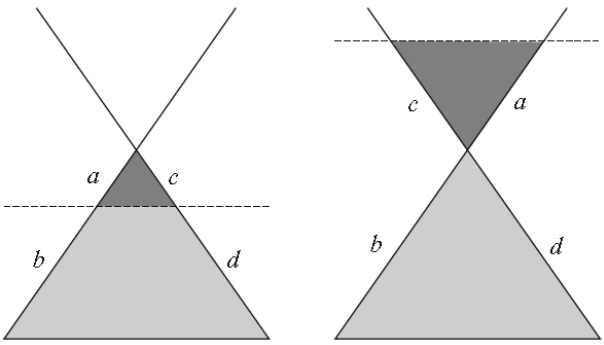


Fig. 7.32 The first intercept theorem.

7.6.1.2 Intercept Theorems

See Figures 7.32 and 7.33, where the diagrams show two intersecting lines and the parallel lines that give rise to the following observations:

First intercept theorem:

$$\frac{a+b}{a}=\frac{c+d}{c}, \frac{b}{a}=\frac{d}{c}.$$

Second intercept theorem:

$$\frac{a}{b}=\frac{c}{d}.$$

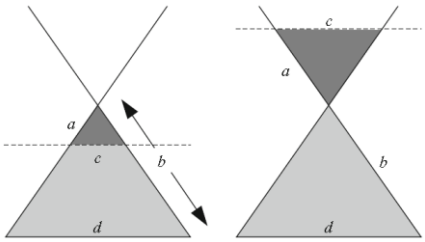


Fig. 7.33 The second intercept theorem.

7.6.1.3 Golden Section

The *golden section* is an 'ideal' ratio for the height and width of an object. Its origins from the interaction between a circle and triangle and give rise to the relationship as given below:

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

The rectangle as shown in Fig. 7.34 has the following proportions:

height = 0.618 ×*width.*

It is interesting to note that the most widely observed rectangle, the television, has no relation to this ratio.

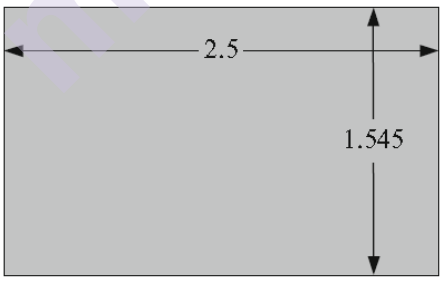


Fig. 7.34 A rectangle with a height to width ratio equal to the golden section.

7.6.1.4 Triangles

The *interior* and *exterior* angles of a triangle has some rules which are very useful in solving all sorts of geometric problems. You can observe Figure 7.35 which shows two diagrams identifying interior and exterior angles. The sum of the interior angles is 180◦, also, the exterior angles of a triangle are equal to the sum of the opposite angles:

$\alpha + \beta + \theta = 180^\circ$

$\alpha = \theta + \beta$

$\beta = \alpha + \theta$

$\theta = \alpha + \beta$.

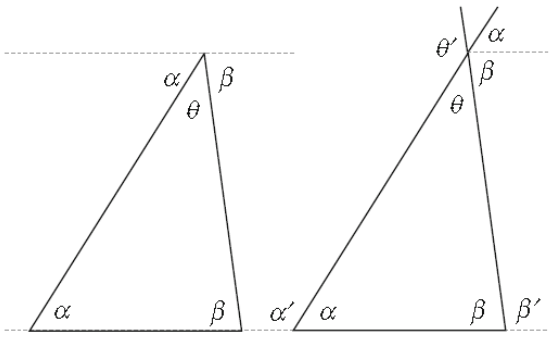


Fig. 7.35 Relationship between interior and exterior angles.

7.6.1.5 Centre of Gravity of a Triangle

A *median* is defined as a straight line joining the vertex of a triangle to the mid-point of the opposite side. If we draw all three medians, they intersect at a common point, which is also the triangle's *center of gravity*. This center of gravity divides all the medians in the ratio 2 : 1. In Figure 7.36 you can observe this.

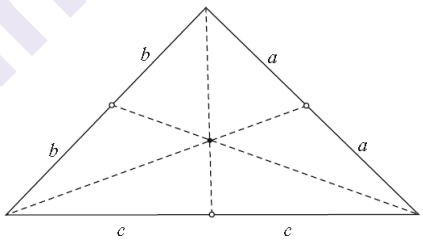


Fig. 7.36 The three medians of a triangle intersect at its center of gravity.

7.6.1.6 Isosceles Triangle

In Figure 7.37 you can see an *isosceles* triangle, it has two equal sides of length *l* and equal base angles α. The triangle's altitude and area are

$$h = \sqrt{l^2 - \left(\frac{c}{2}\right)^2} \qquad A = \frac{ch}{2} .$$

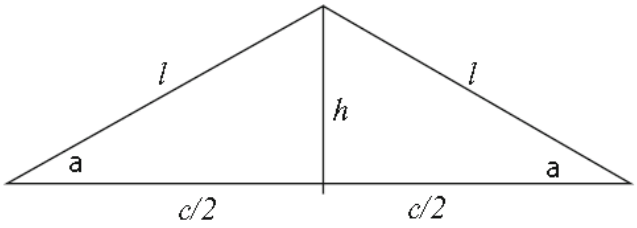


Fig. 7.37 An isosceles triangle.

7.6.1.7 Equilateral Triangle

The *equilateral* triangle possesses three equal sides of length l and equal angles of 60◦. The triangle's altitude and area are

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

7.6.1.8 Right Triangle

Right angle is one famous type which we all know. The Figure 7.38 shows a right triangle with its obligatory right angle. The triangle's altitude and area are

$$h = \frac{ab}{c} \qquad A = \frac{ab}{2}$$

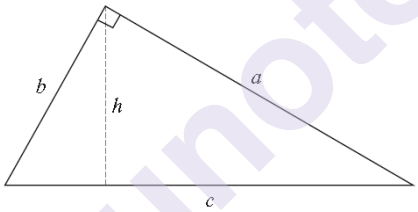


Fig. 7.38 A right triangle.

7.6.1.9 Theorem of Thales

In Figure 7.39 the Theorem of Thales is illustrated, which states that the right angle of a right triangle lies on the circumcircle over the hypotenuse.

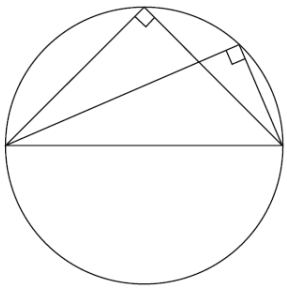


Fig. 7.39 The Theorem of Thales states that the right angle of a right triangle lies on the circumcircle over the hypotenuse.

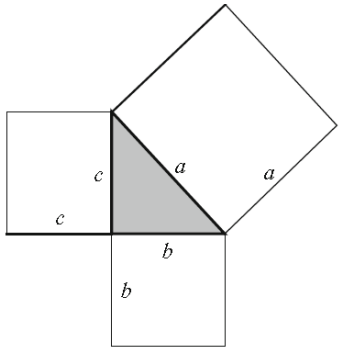


Fig. 7.40 The Theorem of Pythagoras states that $a^2 = b^2 + c^2$.

7.6.1.10 Theorem of Pythagoras

Although the theorem is named after Pythagoras, it was known by the Babylonians a millennium earlier. However, Pythagoras is credited for the proof. In Figure 7.40 you can see well-known relationship

$a^2 = b^2+c^2$

from which one can show that

$\sin^2\alpha + \cos^2\alpha = 1$.

7.6.1.11 Quadrilaterals

Quadrilaterals are known for having four sides and including the square, rectangle, trapezoid, parallelogram and rhombus, whose interior angles sum to 360◦. As the square and rectangle are familiar shapes, we will only consider remaining three.

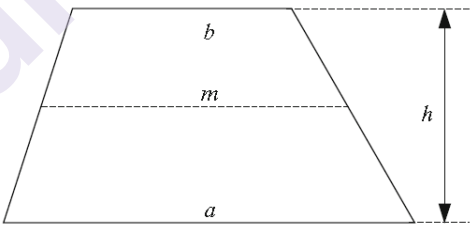


Fig. 7.41 A trapezoid with one pair of parallel sides.

7.6.1.12 Trapezoid

In Figure 7.41 helps to show the *trapezoid* which has one pair of parallel sides *h* apart. The mid-line *m* and area are given by

$$m = \frac{a+b}{2} \qquad A = mh$$

7.6.1.13 Parallelogram

The Figure 7.42 shows the *parallelogram*, which is created from two pairs of intersecting parallel lines, so it has equal opposite sides and equal opposite angles. The altitude, diagonal lengths and area are given by

$$h = b \sin \alpha$$

$$d_{1,2} = \sqrt{a^2 + b^2 \pm 2a\sqrt{b^2 - h^2}}$$

$$A = ah$$

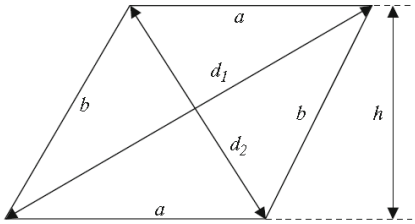


Fig. 7.42 A parallelogram formed by two pairs of parallel lines.

7.6.1.14 Rhombus

Rhombus as shown in Figure 7.43, which is a parallelogram with four sides of equal length *a*. The area is given by

$$A = a^2 \sin \alpha = \frac{d_1 d_2}{2}.$$

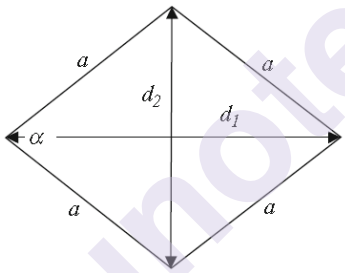


Fig. 7.43 A rhombus is a parallelogram with four equal sides.

7.6.1.15 Regular Polygon (n-gon)

In Figure 7.44 you can see a part of the regular *n*-gon with outer radius R_o, inner radius R_i and edge length a_n.

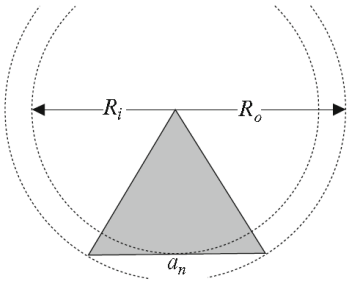


Fig. 7.44 Part of a regular n-gon showing the inner and outer radii and the edge length.

7.6.1.16 Circle

An *annulus* is the area between two concentric circles, and its area *A* is given by

$$A = \pi(R^2 - r^2) = \frac{\pi}{4}(D^2 - d^2)$$

where $D = 2R$ and $d = 2r$.

The area of sector of a circle is given by

$$A = \frac{\alpha^0}{360^0}\pi r^2 .$$

The area of *segment* of a circle is given by

$$A = \frac{r^2}{2}(\alpha - \sin\alpha)$$

7.7 2D ANALYTIC GEOMETRY

Here we will examine familiar descriptions of geometric elements and ways of computing intersections.

7.7.1 Equation of a Straight Line

The well-known equation of a line is

$y = mx+c$

where m is the slope and c the intersection with the y-axis.

You can see this in Fig. 7.45 and this is called the *normal form*.

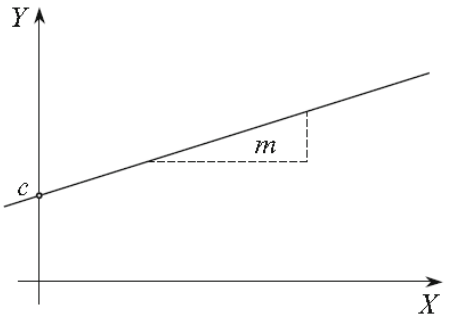


Fig. 7.45 The normal form of the straight line is y = mx+c.

Consider two points (x_1, y_1) and (x_2, y_2) we can state that for any other point (x, y)

$$\frac{y - y_1}{x - x_1} = \frac{y_2 - y_1}{x_2 - x_1}$$

which give

$$y = (x - x_1)\frac{y_2 - y_1}{x_2 - x_1} + y_1 .$$

The more general form is much more convenient:

$ax+by+c = 0$.

7.7.2 The Hessian Normal Form

Consider a line shown in Figure 7.46 whose orientation is controlled by a normal unit vector $\mathbf{n} = [a\ b]^T$. Let $P(x, y)$ is any point on the line, then p is a position vector where $\mathbf{p} = [x\ y]^T$ and d is the perpendicular distance from the origin to the line.

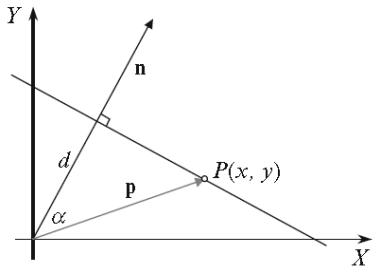


Fig. 7.46 The orientation of a line can be controlled by a normal vector n and a distance d.

Hence,

$$\frac{d}{|P|} = \cos\alpha$$

And

$d = |\mathbf{p}|\cos\alpha$.

But the dot product $\mathbf{n} \cdot \mathbf{p}$ is given by

$\mathbf{n} \cdot \mathbf{p} = |\mathbf{n}||\mathbf{p}|\cos\alpha = ax+by$

we can imply

$ax+by = d|\mathbf{n}|$

and because $|\mathbf{n}| = 1$ we can write

$ax+by-d = 0$

where (x, y) is a point on the line, a and b are the components of a unit vector normal to the line, and d is the perpendicular distance from the origin to the line.

7.7.3 Space Partitioning

The Hessian normal form allows partitioning the space into two zones: the partition that includes the normal vector, and the opposite partition.

Given the equation

$ax+by-d = 0$

Here a point (x, y) on the line satisfies the equation. But if we substitute another point (x_1, y_1) which is in the partition in the direction of the normal vector, it creates the inequality.

$ax_1+by_1-d > 0.$

The point (x_2, y_2) which is in the partition opposite to the direction of the normal vector creates the inequality

$ax_2+by_2-d < 0.$

This space-partitioning feature of the Hessian normal form is useful in clipping lines against polygonal windows.

7.7.4 The Hessian Normal Form from Two Points

Given two points (x_1, y_1) and (x_2, y_2) we can compute the values of a, b and d for the Hessian normal form as follows. To begin, we observe:

$$\frac{y-y_1}{x-x_1} = \frac{y_2-y_1}{x_2-x_1} = \frac{\Delta y}{\Delta x}$$

Hence,

$(y-y_1)\Delta x = (x-x_1)\Delta y$

And also,

$x\Delta y-y\Delta x-(x1\Delta y-y1\Delta x) = 0$ (7.39)

This is the general straight line equation. In Hessian normal form:

$$\sqrt{\Delta x^2+\Delta y^2} = 1 .$$

Hence, the Hessian normal form is given by

$$\frac{x\Delta y-y\Delta x-(x_1\Delta y-y_1\Delta x)}{\sqrt{\Delta x^2+\Delta y^2}} = 0 .$$

7.8 INTERSECTION POINTS

7.8.1 Intersection Point of Two Straight Lines

Given two line equations of the form

$a_1x+b_1y+d_1 = 0$

$a_2x+b_2y+d_2 = 0$

the intersection points are given as,

$$x_i = \frac{b_1d_2 - b_2d_1}{a_1b_2 - a_2b_1}$$

$$y_i = \frac{d_1a_2 - d_2a_1}{a_1b_2 - a_2b_1}$$

We can show using determinants:

$$x_i = \frac{\begin{vmatrix} b_1 & d_1 \\ b_2 & d_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

$$y_i = \frac{\begin{vmatrix} d_1 & a_1 \\ d_2 & a_2 \end{vmatrix}}{\begin{vmatrix} a_1 & b_1 \\ a_2 & b_2 \end{vmatrix}}$$

7.8.2 Intersection Point of Two Line Segments

Line segments in computer graphics represent the edges of shapes and objects.

Consider two line segments given by their end points as ($P_1 – P_2$) and ($P_3 – P_4$). If we locate position vectors to these points, we can write the following vector equations to identify the point of intersection:

$p_i = p_1+t(p_2-p_1)$ (7.40)

$p_i = p_3+s(p_4-p_3)$ (7.41)

where parameters s and t vary between 0 and 1. We can write

$p_1+t(p_2-p_1) = p_3+s(p_4-p_3)$.

Hence

$$s = \frac{(p_1-p_3)+t(p_2-p_1)}{(p_4-p_3)} \quad (7.42)$$

$$t = \frac{(p_3-p_1)+s(p_4-p_3)}{(p_2-p_1)} \quad (7.43)$$

We can write

$$t = \frac{(x_3 - x_1) + s(x_4 - x_3)}{(x_2 - x_1)}$$

$$t = \frac{(y_3 - y_1) + s(y_4 - y_3)}{(y_2 - y_1)}$$

And it gives,

$$s = \frac{x_1(y_3-y_2)+x_2(y_3-y_1)+x_3(y_2-y_1)}{(x_2-x_1)(y_4-y_3)-(x_4-x_3)(y_2-y_1)} \quad (7.44)$$

And

$$t = \frac{x_1(y_4-y_3)+x_3(y_1-y_4)+x_4(y_3-y_1)}{(x_4-x_3)(y_2-y_1)-(x_2-x_1)(y_4-y_3)} \quad (7.45)$$

7.9 POINT INSIDE A TRIANGLE

We can test whether a point is inside, outside or touching a triangle. The first is related to finding the area of a triangle.

7.9.1 Area of a Triangle

Let's declare a triangle formed by the anti-clockwise points (x_1, y_1), (x_2, y_2) and (x_3, y_3)

The area is given as,

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

This can be simplified as,

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

And further simplification can be given as,

$$A = \frac{1}{2}\begin{vmatrix} x_1 & y_1 & 1 \\ x_2 & y_2 & 1 \\ x_3 & y_3 & 1 \end{vmatrix}.$$

the point P_t is inside the triangle (P_1, P_2, P_3).

• If the area of triangle (P_1, P_2, P_t) is positive, P_t must be to the left of the line (P_1, P_2).

• If the area of triangle (P_2, P_3, P_t) is positive, P_t must be to the left of the line (P_2, P_3).

• If the area of triangle (P_3, P_1, P_t) is positive, P_t must be to the left of the line (P_3, P_1).

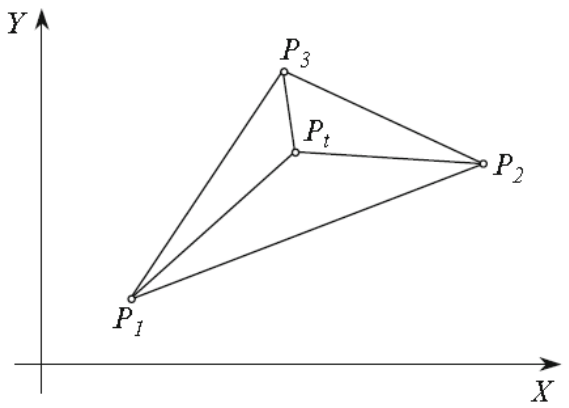


Fig. 7.47 If the point P_t is inside the triangle, it is always to the left as the boundary is traversed in an anti-clockwise direction.

7.9.2 Hessian Normal Form

We can find out of a point is inside, touching or outside a triangle by representing the triangle's edges in the Hessian normal form. If the normal vectors are pointing towards the inside of the triangle, any point which is present inside the triangle will create a positive result when tested against the edge equation for the triangle.

Observe Fig.7.48 which shows a triangle formed by the points (1, 1), (3, 1) and (2, 3).

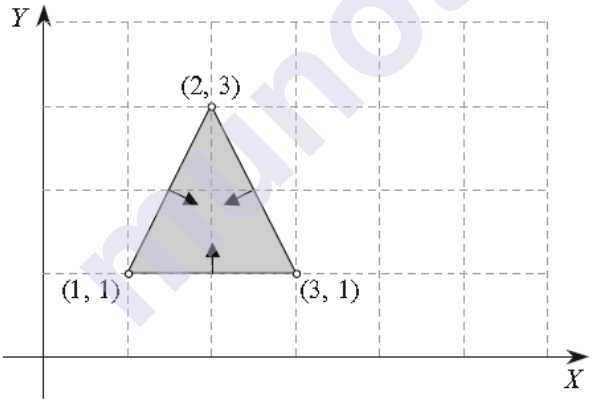


Fig. 7.48 The triangle is represented by three line equations expressed in the Hessian normal form. Any point inside the triangle can be found by evaluating the equations.

We can computer three line equations as follows:

1: The line between (1, 1) and (3, 1):

$0(x-1)+2(1-y) = 0$

$-2y+2 = 0.$

Now multiply this with −1 to reverse the normal vector and get:

$2y-2 = 0$.

2: The line between (3, 1) and (2, 3):

$2(x-3)-1(1-y) = 0$

$2x+y-7 = 0$.

Again multiply by −1 to reverse the normal vector:

$-2x-y+7 = 0$.

3: The last line between (2, 3) and (1, 1):

$-2(x-2)-1(3-y) = 0$.

$-2x+y+1 = 0$.

Finally, multiply by −1 to reverse the normal vector:

$2x-y-1 = 0$.

The three line equations for the triangle are

$2y-2 = 0$

$-2x-y+7 = 0$

$2x-y-1 = 0$.

We are only interested in the sign of the left-hand expressions:

$2y-2$ (7.46)

$-2x-y+7$ (7.47)

$2x-y-1$ (7.48)

This can be tested for any arbitrary point (x, y). And we can conclude that, 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.

7.10 INTERSECTION OF A CIRCLE WITH A STRAIGHT LINE

We have seen the equation of a circle previously, now we will compute its intersection with a straight line. The normal form of line equation is used for testing:

$x2+y2 = r2$ and $y = mx+c$.

By substituting the line equation in the circle's equation, we discover the two intersection points:

$$x_{1,2} = \frac{-mc \pm \sqrt{r^2(1+m^2)-c^2}}{1+m^2} \quad (7.49)$$

$$y_{1,2} = \frac{c \pm m\sqrt{r^2(1+m^2)-c^2}}{1+m^2} \quad (7.50)$$

These points can be used to calculate the required coordinates.

7.11 QUESTIONS:

1. What is trigonometry and trigonometric ratio?
2. Explain sine and cosine rule.
3. What is interpolation?
4. Write a note on linear interpolation.
5. What is trigonometric interpolation?
6. How to interpolate quaternions?
7. Explain Bezier curves.
8. What are B-Splines?
9. Write a short note on 2D analytical geometry.
10. What is intersection point?

REFERENCES

1. Mathematics for Computer Graphics, John Vince, Springer-Verlag Londo
2. Introduction To 3D Game Programming With Directx® 11,Frank D Luna, Mercury Learning and Information
3. **https://conceptartempire.com/polygon-mesh/**

❄❄❄❄❄❄❄

8

INTRODUCTION TO RENDERING ENGINES

Unit Structure :

8.0 OBJECTIVES:

This chapter would make you understand the following concept:

- Rendering Engines
- AR, VR, MR and XR

- Smart Glasses
- HMD

8.1 INTRODUCTION TO RENDERING ENGINES:

- In a software application the rendering engine is the module that is reasonable for generating the graphical output. Basically, the job of a rendering engine is to convert the applications internal model into a series of pixel brightness's that can be displayed by a monitor (or another graphical device e.g: a printer).
- For example, in a 3D game, the rendering engine might take a collection of 3D polygons as inputs (as well as camera and lighting data) and use that to generate 2D images to be outputted to the monitor.
- In a type setting application the rendering engine might take a string a characters and font data (and other assets e.g., images) as inputs and convert them to well formatted image you see on screen or printed on a page.
- Rendering engines are often written to take advantage of features of graphics cards (e.g., highly parallelized matrix operations).
- Programming rendering engines require a strong understanding of geometry.
- Developing a Rendering Engine requires an understanding of how OpenGL and GPU Shaders work.
- Rendering engines is one of the few areas where the effort of code optimization makes sense.

8.2 CURRENT MARKET RENDERING ENGINES

1. 3Delight
2. Arion
3. Arnold
4. Artlantis
5. Clarisse
6. Corona
7. FelixRender
8. FurryBall
9. Guerilla Render
10. Iray
11. Keyshot
12. Blender

- A rendered image can be understood in terms of a number of visible features.
- Rendering research and development has been largely motivated by finding ways to simulate these efficiently. Some relate directly to particular algorithms and techniques, while others are produced together.

8.3 RENDERING FEATURES & TECHNIQUES

1. Shading – how the color and brightness of a surface varies with lighting
2. Texture-mapping – a method of applying detail to surfaces
3. Bump-mapping – a method of simulating small-scale bumpiness on surfaces
4. Fogging/participating medium – how light dims when passing through non-clear atmosphere or air
5. Shadows – the effect of obstructing light
6. Soft shadows – varying darkness caused by partially obscured light sources
7. Reflection – mirror-like or highly glossy reflection
8. Transparency (optics), transparency (graphic) or opacity – sharp transmission of light through solid objects
9. Translucency – highly scattered transmission of light through solid objects
10. Refraction – bending of light associated with transparency
11. Diffraction – bending, spreading, and interference of light passing by an object or aperture that disrupts the ray
12. Indirect illumination – surfaces illuminated by light reflected off other surfaces, rather than directly from a light source (also known as global illumination)
13. Caustics (a form of indirect illumination) – reflection of light off a shiny object, or focusing of light through a transparent object, to produce bright highlights on another object
14. Depth of field – objects appear blurry or out of focus when too far in front of or behind the object in focus
15. Motion blur – objects appear blurry due to high-speed motion, or the motion of the camera
16. Non-photorealistic rendering – rendering of scenes in an artistic style, intended to look like a painting or drawing.

8.4 UNDERSTANDING THE CURRENT MARKET OF RENDERING ENGINE

- There are a lot of varieties in render engines and 3D design software when considering them for a professional use. This has made 3D artists, users as well as render farm tend to look at their functional features when scavenging for the right software. Since this task can be daunting there is no better option than diving towards render engines while considering the following capabilities;

Unlike software, render engines comes with only two general categories;

1. CPU based render engine
2. GPU based render engine

There is also another category known as a hybrid render engine. This model can utilize the power of both the CPU and GPU at the same time.

Generally, render engines have their own ways in which they perform renderings. While some render engines carry out a biased render, others work with an unbiased rendering principle.

Biased Render Engine

- Simply put, biased is term used when all information is put together before the rays are sent to the camera.
- It can be described at the improvement of algorithms to increase the render time.
- This process does not really define light in its physical form but tries to arrive at an approximation of how the lighting should look.
- In quotes; Biased means limiting – you are setting the limit to being realistic.

Some examples of a Biased Render engine include;

- V-Ray
- RedShift
- Mental Ray
- Render Man

From the list above, V-Ray is a hybrid render engine. It is one of those outstanding engines that can render files using the biased and unbiased principle.

Unbiased Render Engine

- Unlike biased render, unbiased rendering means there is no cheating.

- The system already has concrete information that it is sending to the processors.
- In other words, there are no short turns when calculating rays in an unbiased rendering situation.
- Since the engine will need absolute information before proceeding, it makes it to produce outstanding render quality. The only disadvantage here is the rendering speed.
- People usually misquote unbiased render as the rendering that is most accurate physically.
- Looking at the specifics, none of the above rendering methods are accurate. The difference comes in when you look at the use of BRDF like Blinn or GGX as the approximation of the material is real life.

The unbiased render engine is usually used by film industries. And examples of such render engines include;

- Arnold
- Maxwell
- Octane
- Indigo
- Fstorm
- Corona

In the list above, one of the render engines in this category that makes use both the biased and unbiased principle is Corona.

8.5 UNDERSTANDING AUGMENTED REALITY

- Augmented reality (AR) is an enhanced version of the real physical world that is achieved through the use of digital visual elements, sound, or other sensory stimuli delivered via technology. It is a growing trend among companies involved in mobile computing and business applications in particular.
- The most famous example of AR technology is the mobile app Pokemon Go, which was released in 2016 and quickly became an inescapable sensation. In the game, players locate and capture Pokémon characters that pop up in the real world—on your sidewalk, in a fountain, even in your own bathroom.
- Augmented reality (AR) is an interactive experience of a real-world environment where the objects that reside in the real world are enhanced by computer-generated perceptual information, sometimes across multiple sensory modalities, including visual, auditory, haptic.

- AR can be defined as a system that incorporates three basic features: a combination of real and virtual worlds, real-time interaction, and accurate 3D registration of virtual and real objects.
- In simple word we can defined AR as adding information and meaning to real world object.
- It is a combination of real scene viewed by a user and a virtual scene generated by a
- computer that augments the scene with additional information.
- Goal of augmented reality is to add information and meaning to a real object or place.
- It Enables learner to" EXPLORE" the physical world without assuming any prior knowledge.
- It adds audio commentary, location data, historical context or other forms of content that can make a user's experience of a thing or a place more meaningful.
- **Augmented reality (AR)** adds digital elements to a live view often by using the camera on a smartphone.

8.5.1 ADVANTAGES OF AUGMENTED REALITY(AR):

1. Anyone can use it.
2. When used in medical field to train it can save lives.
3. Can be used in exposing military personal to real lives situations without
4. exposing them to the real life danger.
5. Can save millions of dollars by testing situations (like new buildings)to
6. confirm their success.
7. Knowledge information increments are possible.
8. Experiences are shared between people in real time.
9. Video games provide an even more "real" experience.

8.5.2 DISADVANTAGES OF AUGMENTED REALITY(AR):

1. Production is expensive.
2. Augmented reality games like "first person shooters" have been believed to increase teen aggression because they increase violence.
3. Openness: Content layers can be developed by consumers for display.
4. The use of facial recognition technology combined with geo location and augmented data will display your Facebook status , tweets etc.

5. Information overload and augmenting without permission.

8.5.3 Application of AR

1. Medical Training

   From operating MRI equipment to performing complex surgeries, AR technology holds the potential to boost the depth and effectiveness of medical training in many areas.

2. Retail

   In today's physical retail environment, shoppers are using their smartphones more than ever to compare prices or look up additional information on products they're browsing. For example, Users can view a motorcycle they might be interesting in buying in the showroom, and customize it using the app to see which colours and features they might like.

3. Design & Modelling

   From interior design to architecture and construction, AR is helping professionals to visualize their final products during the creative process. Use of headsets enables architects, engineers, and design professionals' step directly into their buildings and spaces to see how their designs might look, and even make virtual on the spot changes. Urban planners can even model how entire city layouts might look using AR headset visualization. Any design or modelling jobs that involve spatial relationships are a perfect use case for AR technology.

4. Classroom Education

   While technology like tablets have become widespread in many schools and classrooms, teachers and educators are now taking up student's learning experience with AR. Students learning about astronomy might see a full map of the solar system, or those in a music class might be able to see musical notes in real time as they learn to play an instrument.

5. Entertainment

   In the entertainment industry, it's all about building a strong relationship with your branded characters and the audience. Entertainment brands are now seeing AR as a great marketing opportunity to build deeper bonds between their characters and audience. As a matter of fact, the makers of AR sensation Pokemon Go are soon planning to release a Harry Potter-themed AR game that fans can interact with day in and day out.

6. Military

   Integrated Visual Augmentation System (IVAS) is being developed by Microsoft in partnership with the US Army to improve soldiers'

situational awareness, comms, battlefield navigation, and overall operational efficiency.

IVAS integrates Microsoft's HoloLens tech and features a heads-up display (HUD), thermal imaging, interactive maps, and overhead compass. With it, soldiers can track and share enemy positions across the board. It can also detect friendly, neutral, and hostile targets.

It allows you to see in the dark (night vision), to see through smoke, and even peek around corners, thanks to multiple front-facing head-mounted cameras. Soldiers can also look back and review ops by watching a video game-like replay of their last operation, among many other features.

8.6 VIRTUAL REALITY

- Virtual Reality is an artificial environment that is created with the software and presented to the user in such a way that the user starts to believe and accept it as a real environment on a computer.
- Virtual reality is a computerized simulation of new spaces. It can be similar to or completely different from the real world.
- **Virtual reality (VR)** implies a complete immersion experience that shuts out the physical world.
- Using VR devices such as HTC Vive, Oculus Rift or Google Cardboard, users can be transported into a number of real-world and imagined environments such as the middle of a squawking penguin colony or even the back of a dragon.

8.6.1 Application of VR

1. Entertainment

   Many video games already have this technology that allows us to improve 3D graphics, immerse the user in history and, above all, facilitate their use with increasingly less intrusive and simple accessories.

2. Education

   It is one of the most extensive fields of use in which technology is useful, whether for college or university. Virtual reality allows from visiting museums at a distance as Google did with the exhibition on Frida Kahlo; to design buildings or learn about constellations and planets.

3. Medicine

   It is also used in the health field, for example, in cases of specific surgery to virtualize and simulate body parts before an operation. Also, for therapies that help treat phobias or traumas.

4. Commercial (Online Shopping & Retail)

Personalized shopping experiences are provided by creating virtual stores like IKEA Reality Kitchen Experience where customer can explore the store with your VR headset and examine different products before completing the purchase without leaving their houses.

The real estate industry has achieved many benefits with virtual reality applications as you can experience a virtual tour to potential listings and check the whole location without the need of physical effort for commuting and checking more and more available listings till choosing one of them to buy.

Reduced costs and higher returns are competitive advantages of using VR for commercial purposes in addition to reaching desired levels of customers' satisfaction.

5. Tourism and Hospitality

Virtual Tours for existing real-life locations is an exciting advantage of virtual reality application as you can motivate potential visitors with a virtual experience of vacation location, museums, landscapes, festivals to book their tickets after experiencing how it would be enjoyable to visit these locations.

Hotels and resorts also can benefit from the advantages of virtual reality by creating a virtual experience of how customers will be served to encourage potential customers to choose your hotel over competitors in addition to training staff with stimulated situations to improve the clients' satisfaction.

8.7 DIFFERENCES BETWEEN AR AND VR

AUGMENTED REALITY(AR)	VIRTUAL REALITY(VR)
Augmented reality enhances real life with artificial images and adds graphics, sounds to the natural world as it exists.	Virtual reality replaces the real world with artificial.
User is not cut from the reality user can interact with the real world and at the same time can see both real and virtual world.	The user enters an entirely immersive world and cut off from the real world.
AR uses device such as smartphone or wearable device which contains software sensors, a compass and small digital projector which display images onto real world objects.	VR might work better for video games and social networking in a virtual environment such as second life or even play station home.

These phones have GPRS which obtains information about a particular geographical location which can be overlaid with tags etc. images, videos etc can be imposed onto this location.	Here the head mounted displays (HMD)&input devices block out all the external world from the viewer and present a view that is under the complete control of the computer

8.8 MIXED REALITY

- Mixed reality is a blend of virtual reality and augmented reality it is also known as hybrid reality.
- Mixed reality is the integration of real and virtual worlds to produce new visualizations, where physical and digital objects co-exist and interact in real-time.
- Mixed reality technology is just now starting to take off with Microsoft's HoloLens one of the most notable early mixed reality apparatuses.

8.8.1 Application of Mixed Reality

1. Education

   Mixed reality technologies are being used within the education industry to both enhance students' ability to learn and take in information. It also gives the students the opportunity to personalize the way they learn.

   Using 3D projections and simulations, students can interact with and manipulate virtual objects in order to study them in a way that is relevant to themselves and their studies. By inserting three-dimensional objects into a classroom as a means of gauging the size, shape, or other features of a defined "virtual" object, students can gain a deeper sense of understanding as to what it is they're studying.

   Some ways that MR can help in the classroom?

   - Interact with the environment in an immersive experience.
   - Touch and manipulate objects.
   - It is an engaging and fun way of learning.
   - MR can teach any kind of subject.

2. Engineering

   Mixed reality in engineering is slowly but surely becoming a game-changer. From 3D modeling and virtual sculpting to remote repair guidance and project monitoring apps. There are various ways in which the engineering sector has begun to take advantage of mixed reality devices.

Some benefits in Engineering?

- Real-time simulation of engineering processes.
- Use MR with an industrial IoT device to monitor services.
- Engineering training.

For example, using 3D modeling apps on mixed reality devices, professionals are able to build their projects up in a shared virtual environment. This type of detailed 3D modeling + collaboration gives engineers the best chance for spotting errors while also allowing real-time manipulation of their designs. The collaboration environment allows supervisors to evaluate and check their 3D designs in real-time.

3. Training Military Personnel

   A battlefield simulation can emerge using this technology. However, it is useful for training military warriors. The real-time experiences will help them to understand ground strategies and implement better.

4. Healthcare

   When it comes to healthcare, mixed reality technologies have many potential applications. The most obvious is training and education. An example is the over the-shoulder surgeries, where surgical students can be taught remotely by experts as they perform surgeries in real-time.

   Another example is interactive learning. Topics like anatomy with mixed reality technology can be used to map the different layers of the human body. Being able to produce three-dimensional models of the anatomy complete with information accessible by just a simple gesture could change the way health care and medicine is taught.

   MR will also transform the way in which medical students learn, using three-dimensional holograms in a virtual environment rather than two-dimensional diagrams from medical textbooks in base reality.

5. Business

   In business, selling a product evolved for many years. Most importantly, the mixed reality will bring enormous shifts towards sales. To clarify, the product catalog helping the customers to choose will change to digital formats.

   This will help us to choose a specific and accurate product. However, this will increase the production of what the customer needs and higher sales.

8.9 INTRODUCTION TO XR

- Extended Reality (XR) refers to all real-and-virtual environments generated by computer graphics and wearables.
- The 'X' in XR is simply a variable that can stand for any letter.

- XR is the umbrella category that covers all the various forms of computer-altered reality, including: Augmented Reality (AR), Mixed Reality (MR), and Virtual Reality (VR).

8.10 CONCEPTUAL DIFFERENCES BETWEEN AR, VR, MR AND XR

It is important to note that these new technologies come from different places and they seek to do different things. Still, they can use some similar technologies. For example, 3D objects and AI are important to all of them. So, let's look at the concepts and definitions hiding under these words.

- **Virtual Reality (VR)**

  Virtual Reality (VR) is an immersive experience also called a computer-simulated reality. It refers to computer technologies using reality headsets to generate the realistic sounds, images and other sensations that replicate a real environment or create an imaginary world. VR is a way to immerse users in an entirely virtual world. A true VR environment will engage all five senses (taste, sight, smell, touch, sound), but it is important to say that this is not always possible.

  Today, it is easy to say that VR is a well-established new reality-tech. Moreover, after years of popularity in the gaming industry, we are now seeing this technology into more practical applications. The market and the industry are still excited about this tech trend and further progress is expected in the near future.

- **Augmented Reality (AR)**

  Augmented Reality (AR) is a live, direct or indirect view of a physical, real-world environment whose elements are augmented (or supplemented) by computer-generated sensory input such as sound, video, graphics or GPS data. As AR exists on top of our own world it provides as much freedom as you are given within your normal life. AR utilizes your existing reality and adds to it utilizing a device of some sort. Mobile and tablets are the most popular mediums of AR

now, through the camera, the apps put an overlay of digital content into the environment. Custom headsets are also being used.

- **Mixed Reality (MR)**

  Mixed Reality (MR), sometimes referred to as hybrid reality, is the merging of real and virtual worlds to produce new environments and visualizations where physical and digital objects co-exist and interact in real time. It means placing new imagery within a real space in such a way that the new imagery is able to interact, to an extent, with what is real in the physical world we know. The key characteristic of MR is that the synthetic content and the real-world content are able to react to each other in real time.

- **Extended Reality (XR)**

  Extended Reality (XR) is a newly added term to the dictionary of the technical words. For now, only a few people are aware of XR. Extended Reality refers to all real-and-virtual combined environments and human-machine interactions generated by computer technology and wearables. Extended Reality includes all its descriptive forms like the Augmented Reality (AR), Virtual Reality (VR), Mixed Reality (MR). In other words, XR can be defined as an umbrella, which brings all three Reality (AR, VR, MR) together under one term, leading to less public confusion. Extended reality provides a wide variety and vast number of levels in the Virtuality of partially sensor inputs to Immersive Virtuality.

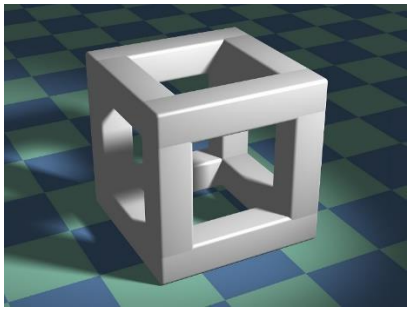
  Since past few years, we have been talking regarding AR, VR, and MR, and probably in coming years, we will be speaking about XR.

8.11 DEPTH MAP

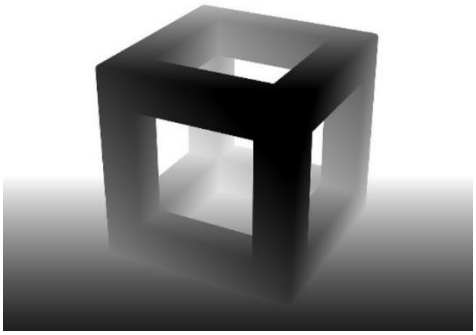
- In 3D computer graphics and computer vision, a depth map is an image or image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint.
- The term is related to and may be analogous to depth buffer, Z-buffer, Z-buffering and Z-depth.
- The "Z" in these latter terms relates to a convention that the central axis of view of a camera is in the direction of the camera's Z axis, and not to the absolute Z axis of a scene.

Examples

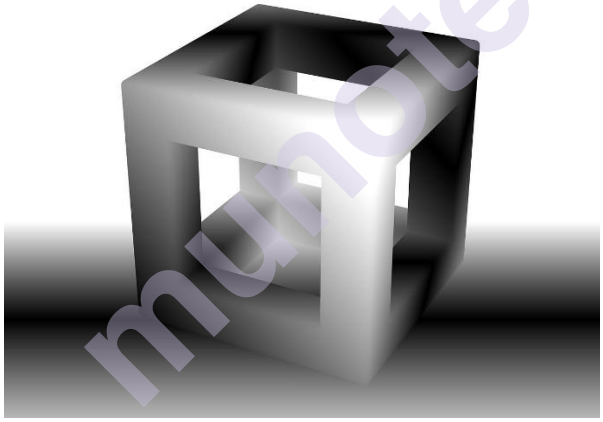
Cubic Structure



Depth Map: Nearer is darker



Depth Map: Nearer the Focal Plane is darker



Two different depth maps can be seen here, together with the original model from which they are derived. The first depth map shows luminance in proportion to the distance from the camera. Nearer surfaces are darker; further surfaces are lighter. The second depth map shows luminance in relation to the distances from a nominal focal plane. Surfaces c loser to the focal plane are darker; surfaces further from the focal plane are lighter, (both closer to and also further away from the viewpoint)

8.12 SMART GLASSES

- Smart glasses are wearable devices that add useful information and functionalities alongside or to what the wearer would normally gather from the real world.

- The added information can be shown visually before your eyes through the display of the glasses or you can get instructions, notifications and answers to your questions in audio form.
- Smart glasses work through a combination of display, sensors and accelerometers, coupled with smart software and internet connectivity to make them really useful.
- They tend to come with touchpads and/or voice controls to help users navigate the software that powers them, which can be embedded into the glasses themselves or incorporated into a handset – or both.

Smart glasses can do a variety of things for you. Among others:

- Send and answer messages and phone calls
- Take photos and videos from your point of view
- Manage your calendar / appointments and get pop-up reminders
- Turn-by-turn GPS navigation
- Interact with apps (Search, fitness tracking, music, Uber, …)
- They also have endless use cases for enterprises. Some examples are:
- Warehouse workers can get real-time information about orders/ inventory and move around with both their hands free
- Manufacture/building companies can display real-time assembly instructions for their employees.
- Medical doctors can record and document patient interaction real time and view medical information from previous visits.

8.12.1 Application of smart glasses

1. Entertainment

   Entertainment, including VR (virtual reality) games, is accessible at any time with smart glasses. Smart glasses will also help you save on other things like for example, a television. Now you can pull up a chair and watch your favourite films in high definition and in 3D right before your eyes, without ever buying a television (or losing the remote control for that matter).

2. Lifelogging

   Let's say you have decided to go for a hike or a holiday somewhere abroad and you want capture and remember every single moment. The solution – consider lifelogging and store all your memories and the sights you have seen by using smart glasses.

3. Voice Commands

   Talking on the phone hands-free while driving may not be a new thing, but answering the call without lifting a finger is.

Voice recognition incorporated into smart glasses can enable you to seamlessly schedule events and notifications, control music, get turn-by-turn navigation and search the web. This list is only scratching the surface – the possibilities are endless.

4. Training

   Smart glasses might just be the thing, to make your training more focused and also entertaining. Listening to music while getting real-time information about the session and measurements from the sensors connected to the glasses can definitely help with your training experience.

5. Facial Recognition

   The smart glasses can also include cool security features such as facial recognition. However, facial recognition has several more impactful applications. For example, the technology is already used in the military and also by the police forces in China where smart glasses can recognize suspicious citizens and travellers in seconds.

8.13 MOBILE PHONE

- Now a day, the mobile phones are extensively using AR, VR, MR.
- The applications of mobile phone with AR, VR and MR are
  1. They are used for gaming
  2. They are used to view how furniture will look in home before buying it.
  3. It can be used to interact with Remote users.
  4. It can be used for seeing menu in plate before making an order.
  5. It can be used in military for Augmentation of battle field scene.

The following are the mobile devices which supports AR,VR and MR:

Company	**VR Phones**
Apple	IPhone 6s , IPhone 6s plus, iphone7, iphone7 plus etc.
Google	Nexus, Pixel, Pixel XL etc
LAVA	Z2 Max,Lava Z6,Lava Z4 etc.
Micromax	Canvas 2 plus,Infinity N11 etc
Sumsung	Galaxy A12,Samsung S9, Samsung S9 Plus etc

8.14 HEAD MOUNTED DEVICE (HMD)

- A Head-mounted Display (HMD) is just what it sounds like -- a computer display you wear on your head. Most HMDs are mounted in a helmet or a set of goggles.
- Engineers designed head-mounted displays to ensure that no matter in what direction a user might look, a monitor would stay in front of his eyes.
- Most HMDs have a screen for each eye, which gives the user the sense that the images he's looking at have depth.
- The monitors in an HMD are most often Liquid Crystal Displays (LCD), though you might come across older models that use Cathode Ray Tube (CRT) displays.
- LCD monitors are more compact, lightweight, efficient and inexpensive than CRT displays. The two major advantages CRT displays have over LCDs are screen resolution and brightness.
- Unfortunately, CRT displays are usually bulky and heavy.
- Almost every HMD using them is either uncomfortable to wear or requires a suspension mechanism to help offset the weight.
- Many head-mounted displays include speakers or headphones so that it can provide both video and audio output.
- The HMD allows viewers to look at an image from various angles or change their field of view by simply moving their heads.
- Major HMD applications include military, government (fire, police, etc.), and civilian-commercial (medicine, video gaming, sports, etc.).

8.15 SUMMARY

- VR is immersing people into a completely virtual environment.
- AR is creating an overlay of virtual content, but can’t interact with the environment.
- MR is a mixed of virtual reality and the reality, it creates virtual objects that can interact with the actual environment.
- XR brings all three Reality (AR, VR, MR) together under one term.
- A depth map is an image or image channel that contains information relating to the distance of the surfaces of scene objects from a viewpoint.
- A head-mounted display (HMD) is a display device, worn on the head or as part of a helmet.

8.16 QUESTIONS:

1. What is Virtual Reality? Explain any two applications of it in detail.
2. What is Augmented Reality? Explain any two applications of it in detail.
3. What is Mixed Reality? Explain any two applications of it in detail.
4. State the difference between VR, AR and MR.
5. Explain the concept of depth mapper.
6. Explain the following with respect to rendering
   a. Mobile phones
   b. Smart classes
   c. HMD's

8.17 REFERENCES

https://www.inc.com/

https://www.computertechreviews.com/

https://www.viget.com/

https://medium.com/

https://en.wikipedia.org/

https://smartglasseshub.com/

https://www.renderboost.com/

❄❄❄❄❄❄❄

9

UNITY ENGINE

Unit Structure :

9.0 OBJECTIVES:

This chapter would make you understand the following concept:

- Unity
- working in Unity
- GameObject
- Component

- Asset
- Prefabs
- Animation in unity
- Timeline in Unity

9.1 INTRODUCTION:

- Unity is a game engine developed by Unity Technologies. It is one of the most widely used engines in the game development industry.
- Since it is a cross-platform engine, it can be used to create games for different platforms like Windows, iOS, Linux, and Android.
- The engine has been adopted by industries outside video gaming, such as film, automotive, architecture, engineering, and construction. As of now, the engine supports as many as 25 platforms.
- It has its own Integrated Development Environment (IDE) and is famous for creating interactive games.
- It contains many elements like Assets, GameObjects, Components, Scenes, and Prefab.
- We Use the Unity Editor to create 2D and 3D games, apps and experiences.

Unity has been used to develop many renowned games like-

- Ghost of a tale
- Firewatch
- Hearthstone- Heroes of warcraft
- Wasteland 2
- Battlestar Galactica Online
- Rust
- Temple Run Trilogy
- Escape plan
- Pokemon Go
- Super Mario Run

9.2 WORKING WITH UNITY

- Unity makes the game production process simple by giving you a set of logical steps to build any conceivable game scenario. Renowned for being non-game-type specific, Unity offers you a blank canvas and a set of consistent procedures to let your imagination be the limit of your creativity.

9.2.1 Essential Unity Concept

Assets

- These are the building blocks of all Unity projects. From graphics in the form of image files, through 3D models and sound files, Unity refers to the files you'll use to create your game as assets.
- This is why in any Unity project folder all files used are stored in a child folder named Assets.

Scenes

- In Unity, you should think of scenes as individual levels, or areas of game content (such as menus).
- By constructing your game with many scenes, you'll be able to distribute loading times and test different parts of your game individually.

Game Objects

- When an asset is used in a game scene, it becomes a new Game Object—referred to in Unity terms—especially in scripting—using the contracted term "GameObject".
- All GameObjects contain at least one component to begin with, that is, the Transform component.
- Transform simply tells the Unity engine the position, rotation, and scale of an object—all described in X, Y, Z coordinate (or in the case of scale, dimensional) order.
- In turn, the component can then be addressed in scripting in order to set an object's position, rotation, or scale. From this initial component, you will build upon game objects with further components adding required functionality to build every part of any game scenario you can imagine.

Components

- Components come in various forms. They can be for creating behavior, defining appearance, and influencing other aspects of an object's function in the game.
- By 'attaching' components to an object, you can immediately apply new parts of the game engine to your object.
- Common components of game production come built-in with Unity, such as the Rigidbody component, down to simpler elements such as lights, cameras, particle emitters, and more.
- To build further interactive elements of the game, you'll write scripts, which are treated as components in Unity.

Scripts

- Unity allows you to create your own Components using scripts. These allow you to trigger game events, modify Component properties over time and respond to user input in any way you like.
- Unity supports the C# programming language natively. C# (pronounced C-sharp) is an industry-standard language similar to Java or C++.
- In addition to this, many other .NET languages can be used with Unity if they can compile a compatible DLL.
- In other words we can say that Script add functionality to a GameObject.

Prefabs

- Prefabs are like blueprints of a GameObject.
- So we can say, Prefabs are a copy of a GameObject that can be duplicated and put into a scene, even if it didn't exist when the scene was being made; in other words, prefabs can be used to generate GameObjects dynamically.

9.2.2 Unity interface

The Unity interface, like many other working environments, has a customizable layout. Consisting of several dockable spaces, you can pick which parts of the interface appear where. Let's take a look at a typical Unity layout:

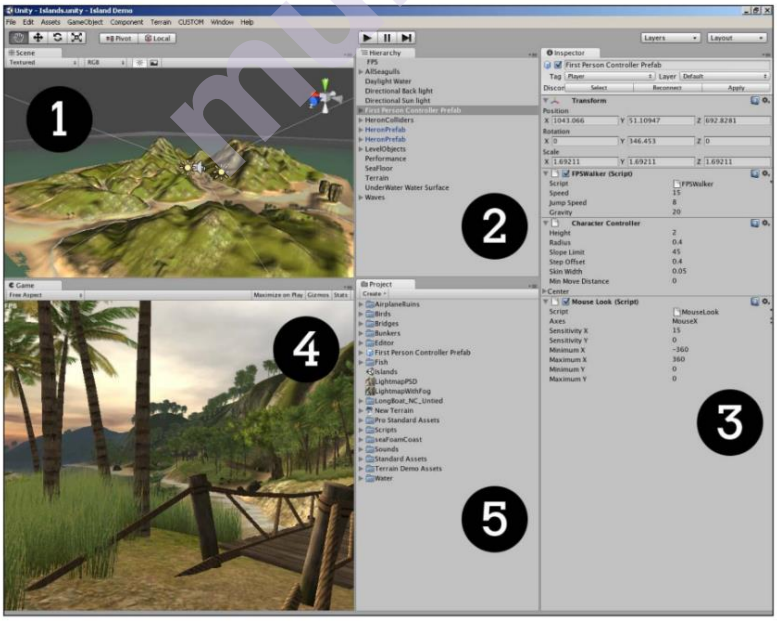
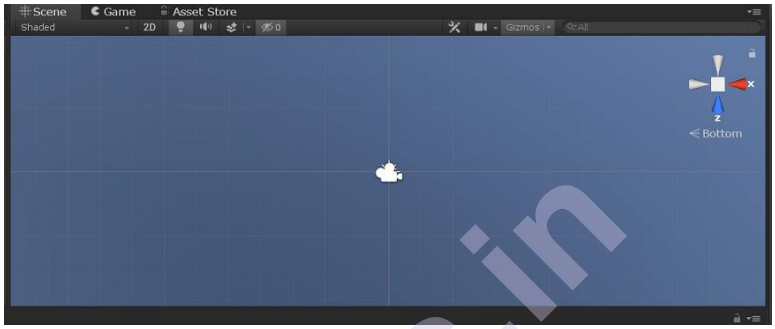


Fig:9.1 Unity Interface

The figure 9.1 image demonstrates that there are five different windows you'll be dealing with:

- Scene [1]—where the game is constructed.
- Hierarchy [2]—a list of GameObjects in the scene.
- Inspector [3]—settings for currently selected asset/object
- Game [4]—the preview window, active only in play mode
- Project [5]—a list of your project's assets, acts as a library

1. Scene View

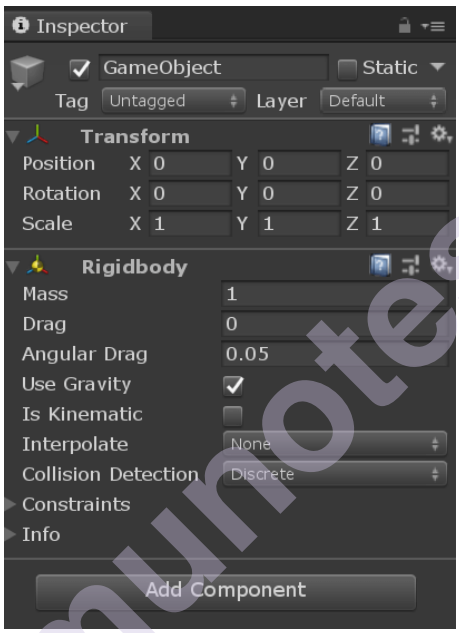


- This window is where we will create our scenes. This view allows you to navigate and edit your scene visually.
- The scene view can show a 2D or 3D perspective, depending on the type of project you are working on.
- We are using the scene view to select and position scenery, cameras, characters, lights, and all other types of GameObject.
- Being able to select, manipulate, and modify objects in the scene view are some of the most important skills you must learn to begin working in Unity.

2. Hierarchy Window

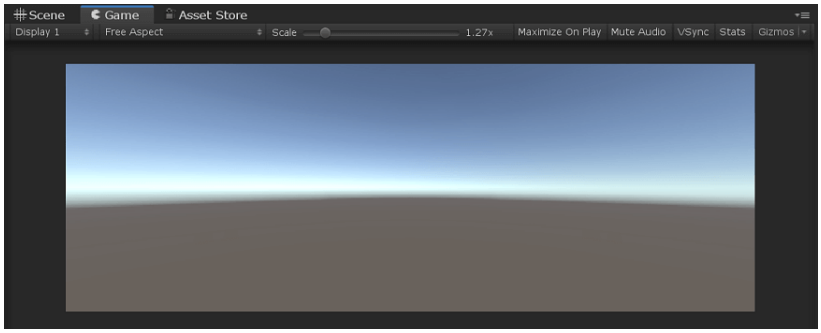
- This is the hierarchy window. This is the hierarchical text representation of every object in the scene. It is where all the objects in your recently open scene are listed, along with their parent-child hierarchy.
- Each item in the scene has an entry in the hierarchy, so the two windows are linked. The hierarchy defines the structure of how objects are attached to one another.
- By default, the Hierarchy window lists GameObjects by order of creation, with the most recently created GameObjects at the bottom. We can reorder the GameObjects by dragging them up or down, or by making the parent or child GameObjects.

3. Inspector Window



- The Inspector window allows you to view and edit all the properties of the currently selected object.
- Since different types of objects have different sets of properties, the layout and contents of the inspector window will vary.
- In this window, you can customize aspects of each element that is in the scene.
- You can select an object in the Hierarchy window or double click on an object in the scene window to show its attributes in the inspector panel.
- The inspector window displays detailed information about the currently selected GameObject, including all attached components and their properties, and allows you to modify the functionality of GameObjects in your scene.

4. Game Window



- This window shows the view that the main camera sees when the game is playing. Means here, you can see a preview window of how the game looks like to the player.
- It is representative of your final game. You will have to use one or more cameras to control what the player actually sees when they are playing your game.

5. Project window

- This window displays the files being used for the game. You can create scripts, folders, etc. by clicking create under the project window.
- In this view, you can access and manage the assets that belong to your project.
- All assets in your project are stored and kept here. All external assets, such as textures, fonts, and sound files, are also kept here before they are used in a scene.
- The favorites section is available above the project structure list. Where you can maintain frequently used items for easy access. You can drag items from the list of project structure to the Favorites and also save search queries there.

9.3 INTRODUCTION TO UNITY 2D

Unity is available for both 2D and 3D games. When you create a new project in Unity, you will have a choice to start in 2D or 3D mode. The choice between starting from 2D or 3D mode determines some settings for the Unity Editor, such as whether images are imported as sprites or textures.

You can swap between 2D or 3D mode at any time regardless of the mode you set when you created your project.

9.3.1 Sprites in Unity

- Sprites are simple 2D graphic objects that have graphical images (called textures) on them. Unity handles sprites by default when the engine is in 2D mode.
- When you view the sprite in 3D space, sprites will appear to be paper-thin, because they have no Z-width.
- Sprites always face the camera at a right angle unless rotated in 3D space

When you create a new sprite, it uses a texture. This texture is then applied on a fresh GameObject, and the Sprite Renderer component is attached to it. This makes our GameObject visible with our texture, as well as its properties related to how it looks on-screen.

9.3.2 Creating Sprites:

To create a sprite to your game, you must supply the engine with a texture. Let's create a texture first.

- Get an image what you want to add as a sprite in standard image file such as PNG or JPG that you want to use,
- Save it in your system directory and
- Then drag the image into the Assets region of Unity.
- Now drag the image from the Assets into the Scene Hierarchy.

You will notice that as soon as you let go of the mouse button, a new GameObject with the name of the texture shows up in the list. You will also get the image now in the middle of the scene in the scene view.

Let us consider the following points while adding a sprite:

- By dragging from an external source into Unity, we are putting an asset.
- This added asset is an image, so it becomes a texture.
- By dragging this texture into the scene hierarchy, we are creating a new GameObject with the same name as our texture, with a sprite renderer attached.
- This sprite renderer uses that texture draws the image in the game.

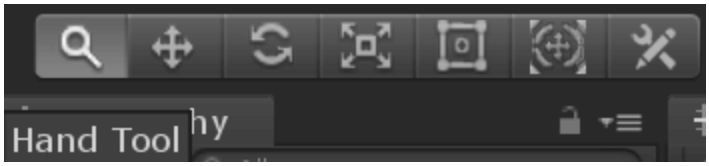
9.3.3 Modifying Sprites

- We can manipulate the imported sprites in various ways to change how it looks.

If you look at the top left corner of the unity interface, you will get a toolbar, as shown below:

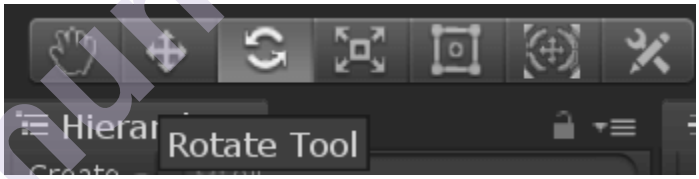
Let's see the functions of these buttons:

A **first-Hand** tool is used to move around the scene without affecting any objects.



The next tool is the **Move** tool. This is used to move the objects in the game world around.

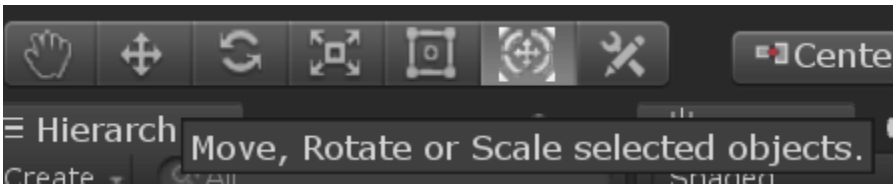
The next tool is the **Rotate** tool, which is used to rotate objects along the Z-axis of the game world or parent object.



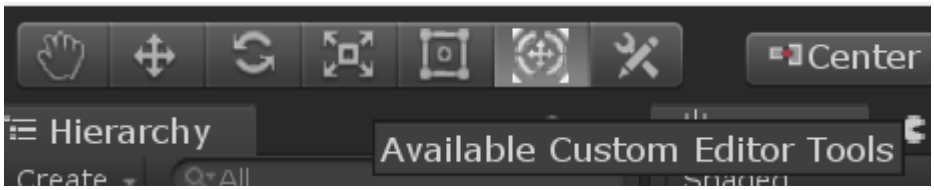
The centered tool is the **Scale** tool. This tool allows you to modify the size (scale) of the objects along certain axes.

The next tool is the **Rect** tool. This tool behaves like a combination of the Move and the Scaling tool but is prone to loss of accuracy. It is more useful in arranging the UI elements.

The next tool is the **Move, Rotate,** and a **Scale** tool. It is used to move, rotate, and scale the selected object.



And finally, the last tool is the **Custom Editor** tool.



These tools are very useful and worthy as the complexity of the project increases.

9.4 GRAPHICS

- Unity's graphics features let you control the appearance of your application and are highly-customizable.
- You can use Unity's graphics features to create beautiful, optimized graphics across a range of platforms, from mobile to high-end consoles and desktop.

9.5 PHYSICS

- Unity helps you simulate physics in your Project to ensure that the objects correctly accelerate and respond to collisions, gravity, and various other forces.
- Unity provides different physics engine implementations which you can use according to your Project needs: 3D, 2D, object-oriented, or data-oriented.
- Physics enables objects to be controlled by (an approximation) of the forces which exist in the real world, such as gravity, velocity and acceleration.

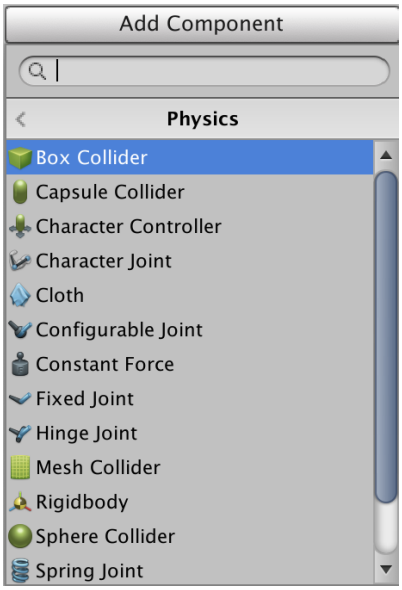


Fig:9.2:Unity Physics Engine Selection

9.5.1 Collider

- Colliders enable Unity to register when GameObjects strike or Intersect each other.
- GameObjects must have a RigidBody component attached to them for collisions to occur.
- Types of colliders include:
    1. Box collider
    2. Capsule collider
    3. Mesh collider
    4. Sphere collider
    5. Wheel collider

Colliders are included in many of Unity's 3D objects from the GameObject dropdown menu. To enable the Unity Physics Engine for a separate or empty game object, click on the Add Component button in the inspector window, select Physics, and specify the type of collider. (Figure 9.2)

9.5.2 Triggers

Enabled via a checkbox on the collider. Functions the same as a collider, but disables physics on the component, enabling objects to pass through it via zone. Events can be called when objects enter or exit the trigger. Figure 9.3

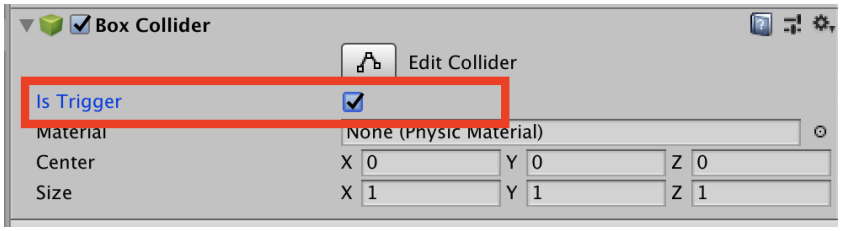


Fig:9.3: Is Trigger checkbox selected in the Box Collider component

One of the objects must have a Rigidbody component attached. As a best practice, objects that move within a Trigger should have this component.

9.5.3 Rigidbody

The Rigidbody component (Figure 9.4) allows GameObjects to be affected by physics properties, such as gravity. It also includes properties of mass, velocity, and air resistance (drag.) Objects of larger mass are less affected by objects with lower mass and vise versa. Drag affects the dampening of velocity over time. Angular Drag affects angular velocity.

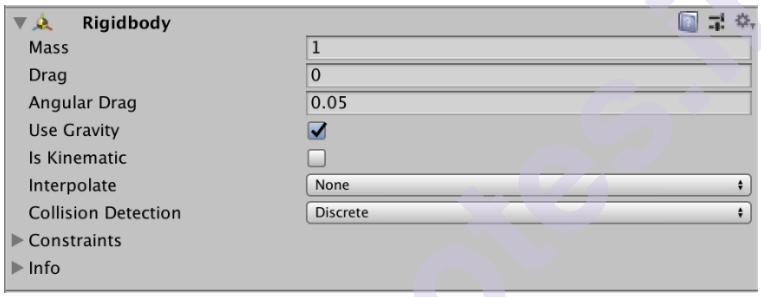


Fig:9.4 The Rigidbody component

The Is Kinematic checkbox allows the Rigidbody to affect other objects via the Unity Physics Engine, but will not be affected themselves. For Example, a Hand Avatar in a VR game can interact with objects via physics, but we don't want physics to act on the hand.

The Is Kinematic checkbox also affects objects controlled by the Animation Engine. If the Is Kinematic checkbox is selected (on),the Animation Engine effects objects. If deselected(off),the physics Engine retains control.

Figure 9.5 shows the default setting for the unity physics engine.

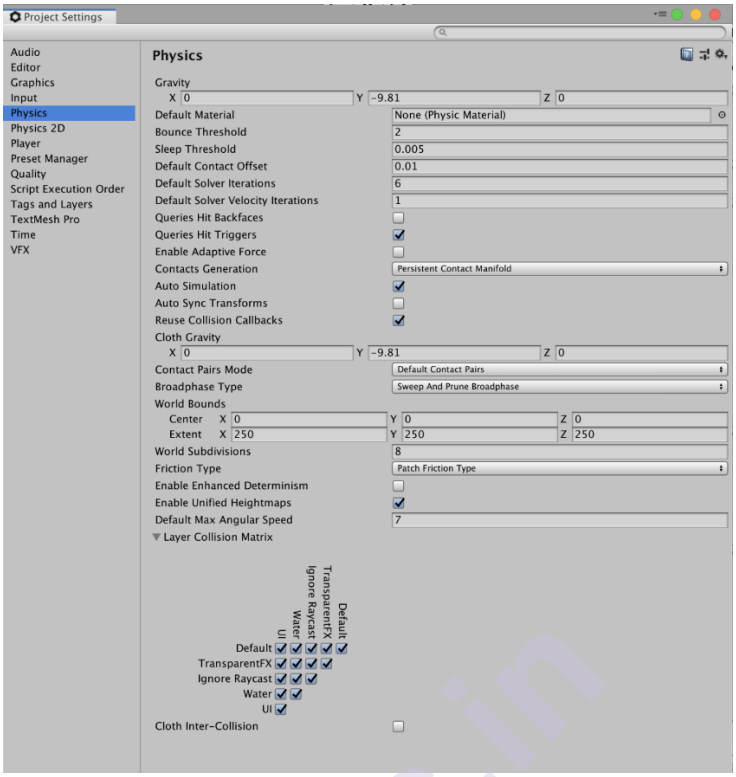


Fig:9.5 Default setting for the unity Physics Engine.

9.6 ANIMATION SYSTEM OVERVIEW IN UNITY

Unity has a rich and sophisticated animation system (sometimes referred to as 'Mecanim').

It provides:

- Easy workflow and setup of animations for all elements of Unity including objects, characters, and properties.
- Support for imported animation clips and animation created within Unity
- Humanoid animation - the ability to apply animations from one character model onto another.
- Simplified workflow for aligning animation clips.
- Convenient preview of animation clips, transitions and interactions between them. This allows animators to work more independently of programmers, prototype and preview their animations before gameplay code is hooked in.
- Management of complex interactions between animations with a visual programming tool.
- Animating different body parts with different logic.

- Layering and masking features.

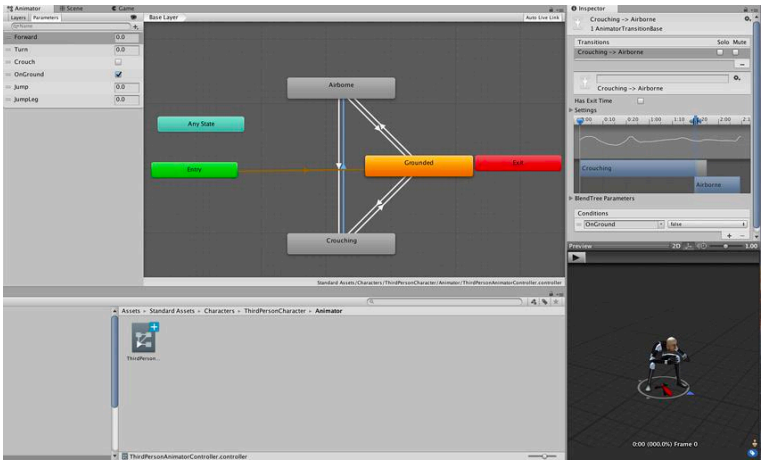


Fig: 9.6 Typical view of an Animation State Machine in the Animator window

9.6.1 Animation workflow

- Unity's animation system is based on the concept of Animation Clips, which contain information about how certain objects should change their position, rotation, or other properties over time. Each clip can be thought of as a single linear recording. Animation clips from external sources are created by artists or animators with 3rd party tools such as Autodesk® 3ds Max® or Autodesk® Maya®, or come from motion capture studios or other sources.

- Animation Clips are then organised into a structured flowchart-like system called an Animator Controller. The Animator Controller acts as a "State Machine" which keeps track of which clip should currently be playing, and when the animations should change or blend together.

- A very simple Animator Controller might only contain one or two clips, for example to control a powerup spinning and bouncing, or to animate a door opening and closing at the correct time. A more advanced Animator Controller might contain dozens of humanoid animations for all the main character's actions, and might blend between multiple clips at the same time to provide a fluid motion as the player moves around the scene.

- Unity's Animation system also has numerous special features for handling humanoid characters which give you the ability to retarget humanoid animation from any source (for example: motion capture; the Asset Store; or some other third-party animation library) to your own character model, as well as adjusting muscle definitions. These special features are enabled by Unity's Avatar system, where humanoid characters are mapped to a common internal format.

- Each of these pieces - the Animation Clips, the Animator Controller, and the Avatar, are brought together on a GameObject via the Animator Component.

- This component has a reference to an Animator Controller, and (if required) the Avatar for this model. The Animator Controller, in turn, contains the references to the Animation Clips it uses.

9.6.2 How the various parts of the animation system connect together

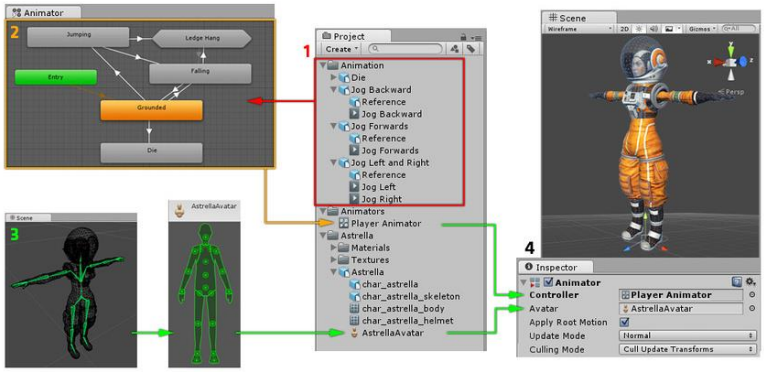


Fig:9.7 Various parts of the animation system connected together

The above Figure 9.7 shows the following:

1. Animation clips are imported from an external source or created within Unity. In this example, they are imported motion captured humanoid animations.

2. The animation clips are placed and arranged in an Animator Controller. This shows a view of an Animator Controller in the Animator window. The States (which may represent animations or nested sub-state machines) appear as nodes connected by lines. This Animator Controller exists as an asset in the Project window.

3. The rigged character model (in this case, the astronaut "Astrella") has a specific configuration of bones which are mapped to Unity's common Avatar format. This mapping is stored as an Avatar asset as part of the imported character model, and also appears in the Project window as shown.

4. When animating the character model, it has an Animator component attached. In the Inspector view shown above, you can see the Animator Component which has both the Animator Controller and the Avatar assigned. The animator uses these together to animate the model. The Avatar reference is only necessary when animating a humanoid character. For other types of animation, only an Animator Controller is required.

9.7 TIMELINE IN UNITY

- Use the Timeline Editor window to create cut-scenes, cinematics, and game-play sequences by visually arranging tracks and clips linked to GameObjects in your scene.

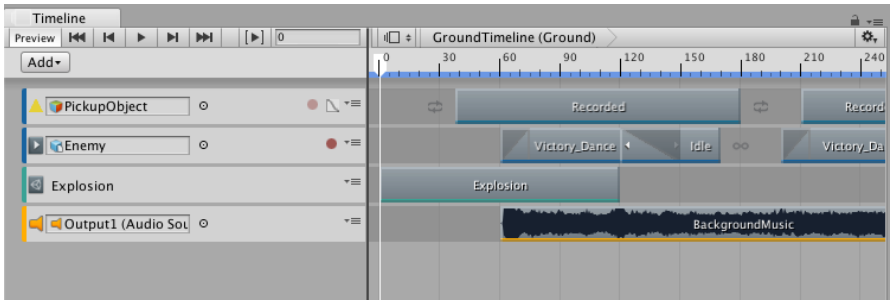


Fig:9.8 A cinematic in the Timeline Editor window.

For each cut-scene, cinematic, or game-play sequence, the Timeline Editor window saves the following:

- **Timeline Asset**: stores the tracks, clips, and recorded animations without links to the specific GameObjects being animated. The Timeline Asset is saved to the project.
- **Timeline instance**: stores links to the specific GameObjects being animated by the Timeline Asset. These links, referred to as bindings, are saved to the scene.
- **Timeline Asset**

  The Timeline Editor window saves track and clip definitions as a Timeline Asset.
- If you record key animations while creating your cinematic, cut-scene, or game-play sequence, the Timeline Editor window saves the recorded animation as children of the Timeline Asset.

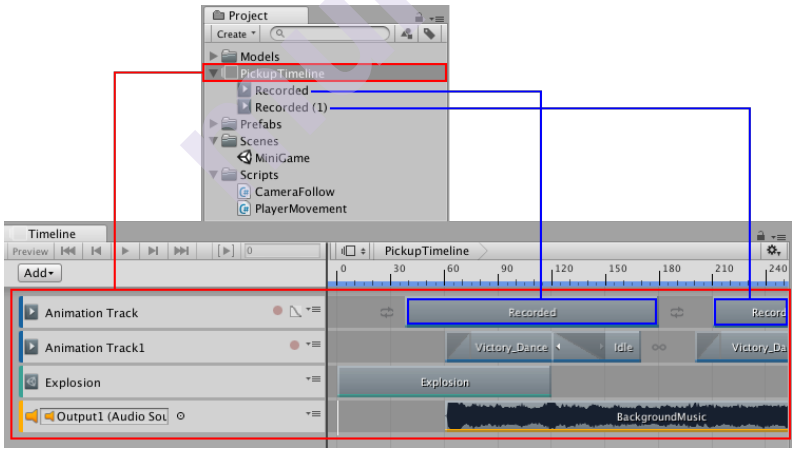


Fig: 9.9 Timeline Asset saves tracks and clips (red). If your record key animation, the recorded clips are saved as children of the Timeline Asset (blue).

Timeline instance

- Although a Timeline Asset defines the tracks and clips for a cut-scene, cinematic, or game-play sequence, you cannot add a Timeline Asset directly to a scene.

- To animate GameObjects in your scene with a Timeline Asset, you must create a Timeline instance.
- The Timeline Editor window provides an automated method of creating a Timeline instance while creating a Timeline Asset.

If you select a GameObject in the scene that has a Playable Director component associated with a Timeline Asset, the bindings appear in the Timeline Editor window and in the Playable Director component (Inspector window).

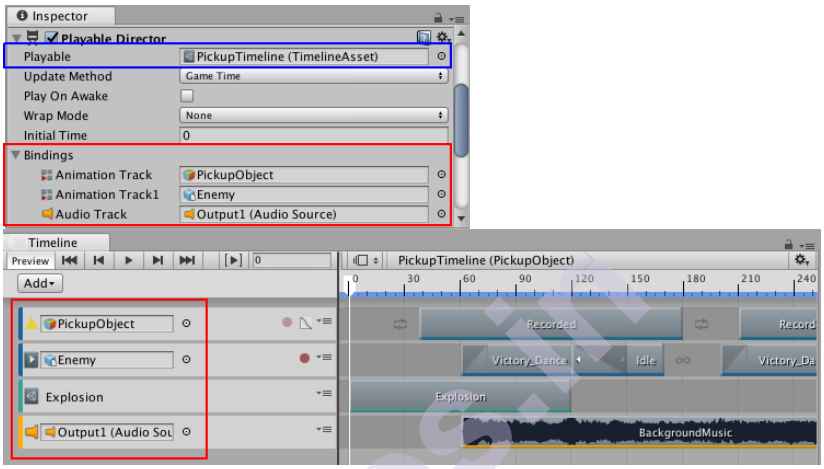


Fig:9.10 The Playable Director component shows the Timeline Asset (blue) with its bound GameObjects (red). The Timeline Editor window shows the same bindings (red) in the Track list.

Reusing Timeline Assets

- Since Timeline Assets and Timeline instances are separate, it is possible to reuse the same Timeline Asset with many Timeline instances.
- For example, you can create a Timeline Asset named VictoryTimeline with the animation, music, and particle effects that play when the main game character (Player) is victorious.
- To reuse the VictoryTimeline Timeline Asset to animate another game character (Enemy) in the same scene, you can create another Timeline instance for the secondary game character.

9.8 WHAT'S THE DIFFERENCE BETWEEN THE ANIMATION WINDOW AND THE TIMELINE WINDOW?

The Timeline window

- The Timeline window allows you to create cinematic content, game-play sequences, audio sequences and complex particle effects.

- You can animate many different GameObjects within the same sequence, such as a cut scene or scripted sequence where a character interacts with scenery.
- In the timeline window you can have multiple types of track, and each track can contain multiple clips that can be moved, trimmed, and blended between.
- It is useful for creating more complex animated sequences that require many different GameObjects to be choreographed together.
- The Timeline window is newer than the Animation window.
- It was added to Unity in version 2017.1, and supercedes some of the functionality of the Animation window.

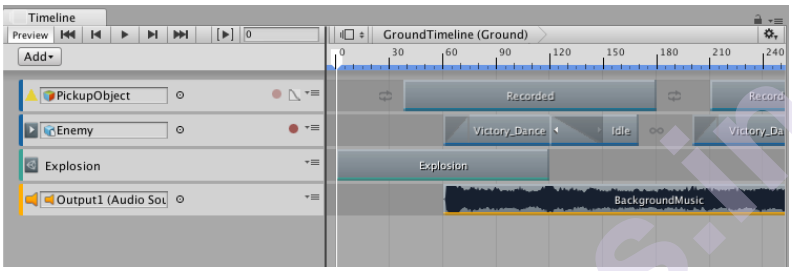


Fig:9.11 Timeline window, showing many different types of clips arranged in the same sequence

The Animation window

- The Animation window allows you to create individual animation clips as well as viewing imported animation clips.
- Animation clips store animation for a single GameObject or a single hierarchy of GameObjects.
- The Animation window is useful for animating discrete items in your game such as a swinging pendulum, a sliding door, or a spinning coin.
- The animation window can only show one animation clip at a time.
- The Animation window was added to Unity in version 4.0.
- The Animation window is an older feature than the Timeline window.
- It provides a simple way to create animation clips and animate individual GameObjects.
- However, to create more complex sequences involving many disparate GameObjects you should use the Timeline window.
- The animation window has a “timeline” as part of its user interface (the horiontal bar with time delineations marked out), however this is separate to the Timeline window.

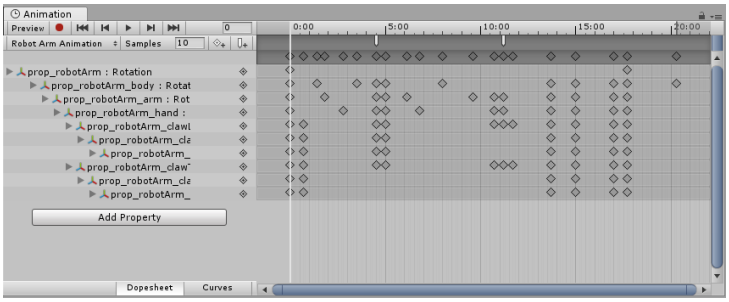


Fig:9.12 Animation Window

9.9 SUMMARY

- Unity is a game engine developed by Unity Technologies. It is one of the most widely used engines in the game development industry.
- Assets are the building blocks of all Unity projects.
- Scripts add functionality to a GameObject.
- Prefabs are like blueprints of a GameObject.
- Unity is available for both 2D and 3D games.
- Physics enables objects to be controlled by (an approximation) of the forces which exist in the real world, such as gravity, velocity and acceleration.
- Colliders enable Unity to register when GameObjects strike or Intersect each other.
- Unity has a rich and sophisticated animation system.
- The Animation window allows you to create individual animation clips as well as viewing imported animation clips.
- The Timeline window allows you to create cinematic content, game-play sequences, audio sequences and complex particle effects.

9.10 QUESTIONS

1) Write a short note on Unity rendering engine.
2) Explain in detail about physics 2D.
3) Write a note on animation window.
4) Explain various unity essential component.
5) Explain timeline window for animation.

9.11 REFERENCES

Unity Game Development Essentials Will Goldstone

https://docs.unity3d.com/

❄❄❄❄❄❄❄

10

SCRIPTING

Unit Structure :

10.0 OBJECTIVE:

This chapter would make you understand the following concept:

- Scripting
- Setting up Multiplayer project
- Navigation and path finding
- Unity Interface

10.1 INTRODUCTION TO SCRIPTING

- Scripting is an essential ingredient in all applications you make in Unity.

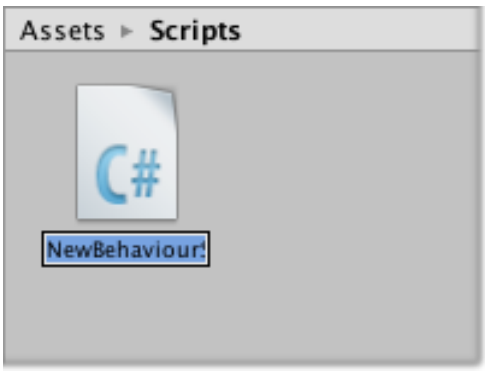
- Most applications need scripts to respond to input from the player and to arrange for events in the gameplay to happen when they should.
- Beyond that, scripts can be used to create graphical effects, control the physical behaviour of objects or even implement a custom AI system for characters in the game.
- Scripting is the process of writing blocks of code that are attached like components to GameObjects in the scene.

10.2 CREATING AND USING SCRIPTS

- The behavior of GameObjects is controlled by the Components that are attached to them. Although Unity's built-in Components can be very versatile.
- Unity allows you to create your own Components using scripts. These allow you to trigger game events, modify Component properties over time and respond to user input in any way you like.
- Unity supports the C# programming language natively. C# (pronounced C-sharp) is an industry-standard language similar to Java or C++.
- In addition to this, many other .NET languages can be used with Unity if they can compile a compatible DLL.

10.2.1 Creating Scripts

- Unlike most other assets, scripts are usually created within Unity directly. You can create a new script from the Create menu at the top left of the Project panel or by selecting Assets > Create > C# Script from the main menu.
- The new script will be created in whichever folder you have selected in the Project panel. The new script file's name will be selected, prompting you to enter a new name.



- It is a good idea to enter the name of the new script at this point rather than editing it later. The name that you enter will be used to create the initial text inside the file.

10.2.2 Anatomy of a Script file

- When you double-click a script Asset in Unity, it will be opened in a text editor. By default, Unity will use Visual Studio, but you can select any editor you like from the External Tools panel in Unity's preferences (go to Unity > Preferences).
- The initial contents of the file will look something like this:

```
using UnityEngine;

using System.Collections;

public class MainPlayer : MonoBehaviour {

// Use this for initialization

void Start () {

   }

   // Update is called once per frame

   void Update ()

 {

   }

}
```

- A script makes its connection with the internal workings of Unity by implementing a class which derives from the built-in class called MonoBehaviour.
- You can think of a class as a kind of blueprint for creating a new Component type that can be attached to GameObjects.
- Each time you attach a script component to a GameObject, it creates a new instance of the object defined by the blueprint.
- The name of the class is taken from the name you supplied when the file was created. The class name and file name must be the same to enable the script component to be attached to a GameObject.
- The main things to note, however, are the two functions defined inside the class.
- The Update function is the place to put code that will handle the frame update for the GameObject.
- This might include movement, triggering actions and responding to user input, basically anything that needs to be handled over time during gameplay.

- To enable the Update function to do its work, it is often useful to be able to set up variables, read preferences and make connections with other GameObjects before any game action takes place.
- The Start function will be called by Unity before gameplay begins (ie, before the Update function is called for the first time) and is an ideal place to do any initialization.
- The construction of objects is handled by the editor and does not take place at the start of gameplay as you might expect. If you attempt to define a constructor for a script component, it will interfere with the normal operation of Unity and can cause major problems with the project.

10.2.3 Controlling a GameObject

- As noted above, a script only defines a blueprint for a Component and so none of its code will be activated until an instance of the script is attached to a GameObject.
- You can attach a script by dragging the script asset to a GameObject in the hierarchy panel or to the inspector of the GameObject that is currently selected.
- There is also a Scripts submenu on the Component menu which will contain all the scripts available in the project, including those you have created yourself. The script instance looks much like any other Component in the Inspector:

- Once attached, the script will start working when you press Play and run the game. You can check this by adding the following code in the Start function:-

```
// Use this for initialization

void Start ()

{

 Debug.Log("I am alive!");

}
```

10.2.4 MonoBehaviour Class

- MonoBehaviour is the base class from which every Unity script derives. When you use C#, you must explicitly derive from MonoBehaviour.

- This class doesn't support the null-conditional operator (?.) and the null-coalescing operator (??).
- The functions in this class are:
- Start() - Start is called on the frame when a script is enabled just before any of the Update methods are called the first time. Start is called exactly once in the lifetime of the script.
- Update() - Update is called every frame, if the MonoBehaviour is enabled. Unity calls this method 60 time per second(i.e 60 frames per second). Not every MonoBehaviour script needs Update.
- FixedUpdate() - The FixedUpdate frequency is more or less than Update. If the application runs at 25 frames per second (fps), Unity calls it approximately twice per frame, Alternatively, 100 fps causes approximately two rendering frames with one FixedUpdate. Use FixedUpdate when using Rigidbody. Set a force to a Rigidbody and it applies each fixed frame. FixedUpdate occurs at a measured time step that typically does not coincide with MonoBehaviour.Update.
- LateUpdate() - LateUpdate is called every frame, if the Behaviour is enabled. LateUpdate is called after all Update functions have been called. This is useful to order script execution. For example a follow camera should always be implemented in LateUpdate because it tracks objects that might have moved inside Update.
- OnGUI() - OnGUI is called for rendering and handling GUI events.
- OnDisable() - This function is called when the behaviour becomes disabled.
- OnEnable() - This function is called when the object becomes enabled and active.

10.3 SETTING UP A MULTIPLAYER PROJECT

- The most basic and common things you need when setting up a multiplayer project. In terms of what you require in your project, these are:

    1) A Network Manager
    2) A user interface (for players to find and join games)
    3) Networked Player Prefabs (for players to control)
    4) Scripts and GameObjects which are multiplayer-aware

- There are variations on this list; for example, in a multiplayer chess game, or a real-time strategy (RTS) game, you don’t need a visible GameObject to represent the player. However, you might still want

an invisible empty GameObject to represent the player, and attach scripts to it which relate to what the player is able to do.

- There are also some important concepts that you need to understand and make choices about when building your game. These concepts can broadly be summarised as:

The Network Manager

- The Network Manager is responsible for managing the networking aspects of your multiplayer game. You should have one (and only one) Network Manager active in your Scene at a time.

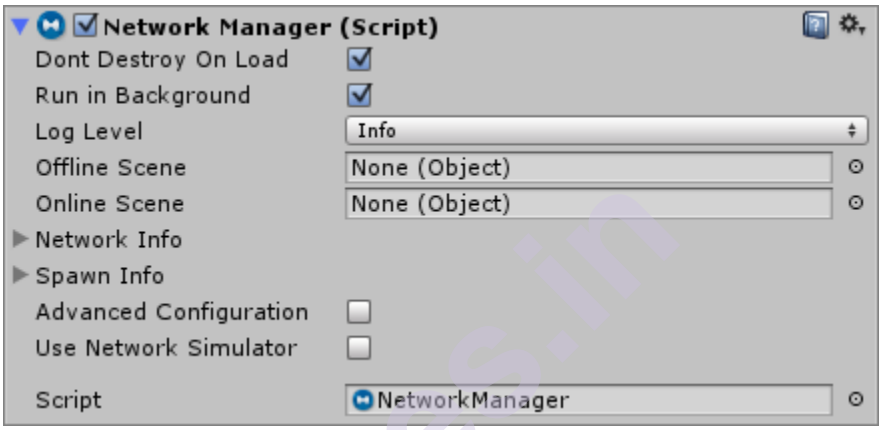


Fig:10.1: The Network Manager Component

- Unity's built-in Network Manager component wraps up all of the features for managing your multiplayer game into one single component. If you have custom requirements which aren't covered by this component, you can write your own network manager in script instead of using this component. If you're just starting out with multiplayer games, you should use this component.

A user interface for players to find and join games

- Almost every multiplayer game provides players with a way to discover, create, and join individual game "instances" (also known as "matches"). This part of the game is commonly known as the "lobby", and sometimes has extra features like chat.

- Unity has an extremely basic built-in version of such an interface, called the NetworkManagerHUD.

- It can be extremely useful in the early stages of creating your game, because it allows you to easily create matches and test your game without needing to implement your own UI.

- However, it is very basic in both functionality and visual design, so you should replace this with your own UI before you finish your project.

Fig:10.2 Unity's built-in Network Manager HUD, shown in MatchMaker mode.

Networked player GameObjects

- Most multiplayer games feature some kind of object that a player can control, like a character, a car, or something else.
- Some multiplayer games don't feature a single visible "player object" but instead allow a player to control many units or items, like in chess or real-time strategy games.
- Others don't even feature specific objects at all, like a shared-canvas painting game.
- In all of these situations, however, you usually need to create a GameObject that conceptually represents the player in your game. Make this GameObject a Prefab, and attach all the scripts to it which control what the player can do in your game.

If you are using Unity's Network Manager component, assign the Prefab to the Player Prefab field.

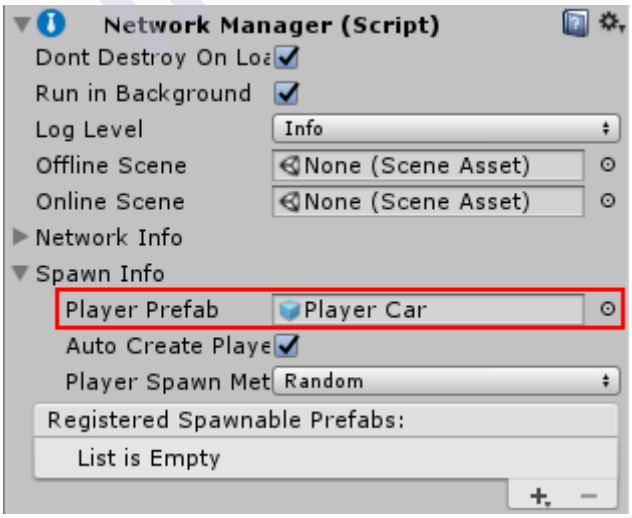


Fig:10.3 The network manager with a "Player Car" prefab assigned to the Player Prefab field.

- When the game is running, the Network Manager creates a copy (an "instance") of your player Prefab for each player that connects to the match.
- However - and this is where it can get confusing for people new to multiplayer programming - you need to make sure the scripts on your player Prefab instance are "aware" of whether the player controlling the instance is using the host computer (the computer that is managing the game) or a client computer (a different computer to the one that is managing the game).
- This is because both situations will be occurring at the same time.

10.4 NAVIGATION AND PATH FINDING

- The navigation system allows you to create characters that can intelligently move around the game world, using navigation meshes that are created automatically from your Scene geometry.
- Dynamic obstacles allow you to alter the navigation of the characters at runtime, while off-mesh links let you build specific actions like opening doors or jumping down from a ledge.
- The Navigation System allows you to create characters which can navigate the game world. It gives your characters the ability to understand that they need to take stairs to reach second floor, or to jump to get over a ditch.
- The Unity **NavMesh** system consists of the following pieces:

1. **NavMesh** (short for Navigation Mesh) is a data structure which describes the walkable surfaces of the game world and allows to find path from one walkable location to another in the game world. The data structure is built, or baked, automatically from your level geometry.
2. **NavMesh Agent** component help you to create characters which avoid each other while moving towards their goal. Agents reason about the game world using the NavMesh and they know how to avoid each other as well as moving obstacles.
3. **Off-Mesh Link** component allows you to incorporate navigation shortcuts which cannot be represented using a walkable surface. For example, jumping over a ditch or a fence, or opening a door before walking through it, can be all described as Off-mesh links.
4. **NavMesh Obstacle** component allows you to describe moving obstacles the agents should avoid while navigating the world. A barrel or a crate controlled by the physics system is a good example of an obstacle. While the obstacle is moving, the agents do their best to avoid it, but once the obstacle becomes stationary it will carve a hole in the navmesh so that the agents can change their paths to steer

around it, or if the stationary obstacle is blocking the path way, the agents can find a different route.

10.5 CREATING USER INTERFACES (UI)

- Unity provides three UI systems that you can use to create user interfaces (UI) for the Unity Editor and applications made in the Unity Editor:

    1. UI Toolkit
    2. The Unity UI package (uGUI)
    3. IMGUI

UI Toolkit

- UI Toolkit is the newest UI system in Unity. It's designed to optimize performance across platforms, and is based on standard web technologies. You can use UI Toolkit to create extensions for the Unity Editor, and to create runtime UI for games and applications (when you install the UI Toolkit package.

UI Toolkit includes:

- A retained-mode UI system that contains the core features and functionality required to create user interfaces.
- UI Asset types inspired by standard web formats such as HTML, XML, and CSS. Use them to structure and style UI.
- Tools and resources for learning to use UI Toolkit, and for creating and debugging your interfaces.
- Unity intends for UI Toolkit to become the recommended UI system for new UI development projects, but it is still missing some features found in Unity UI (uGUI) and IMGUI.

The Unity UI (uGUI) package

- The Unity User Interface (Unity UI) package (also called uGUI) is an older,GameObject-based UI system that you can use to develop runtime UI for games and applications. In Unity UI, you use components and the Game view to arrange, position, and style the user interface. It supports advanced rendering and text features.

IMGUI

- Immediate Mode Graphical User Interface (IMGUI) is a code-driven UI Toolkit that uses the OnGUI function, and scripts that implement it, to draw and manage user interfaces. You can use IMGUI to create custom Inspectors for script components, extensions for the Unity

Editor, and in-game debugging displays. It is not recommended for building runtime UI.

10.5.1 Unity UI: Unity User Interface

Unity UI is a UI toolkit for developing user interfaces for games and applications. It is a GameObject-based UI system that uses Components and the Game View to arrange, position, and style user interfaces.You cannot use Unity UI to create or change user interfaces in the Unity Editor.

Canvas

- The Canvas is the area that all UI elements should be inside. The Canvas is a Game Object with a Canvas component on it, and all UI elements must be children of such a Canvas.
- Creating a new UI element, such as an Image using the menu GameObject > UI > Image, automatically creates a Canvas, if there isn't already a Canvas in the scene.
- The UI element is created as a child to this Canvas.
- The Canvas area is shown as a rectangle in the Scene View. This makes it easy to position UI elements without needing to have the Game View visible at all times.
- Canvas uses the EventSystem object to help the Messaging System.

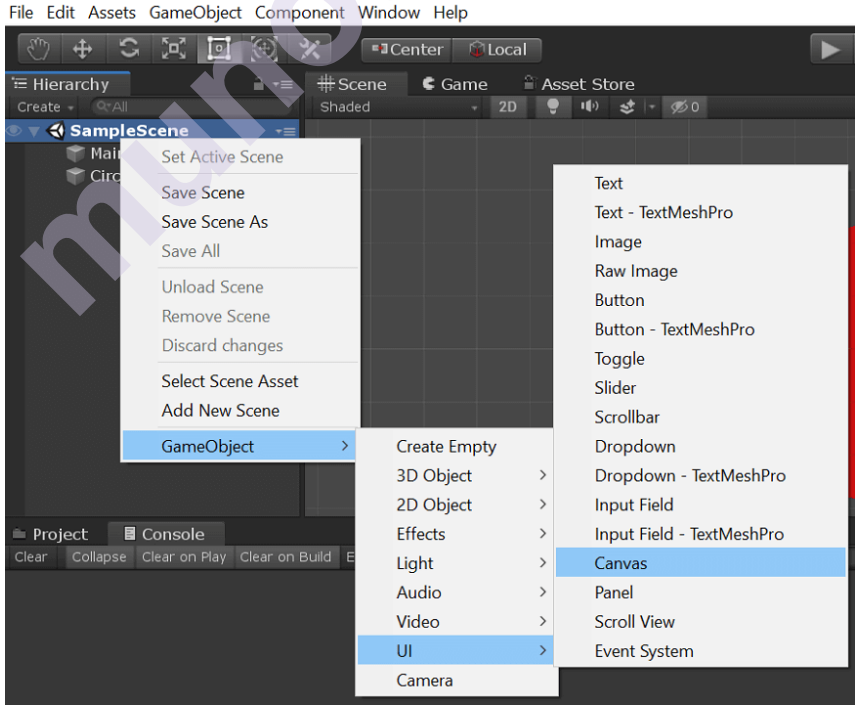
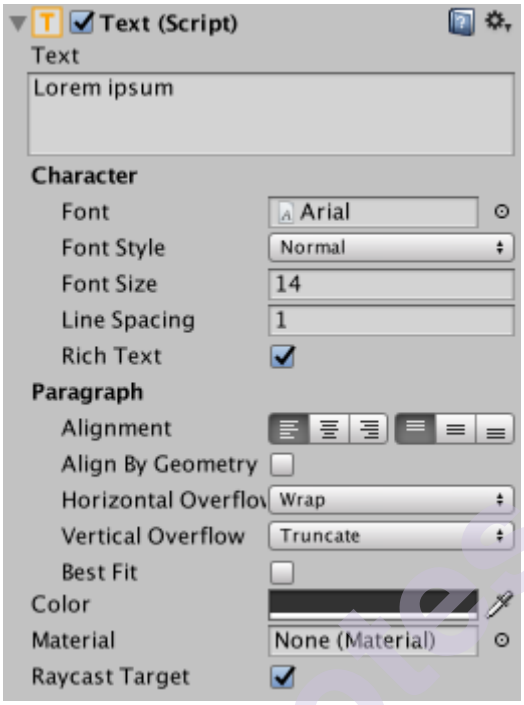


Fig: 10.4 Unity UI: Unity User Interface

With the introduction of the UI system, new Components have been added that will help you create GUI specific functionality. This section will cover the basics of the new Components that can be created.

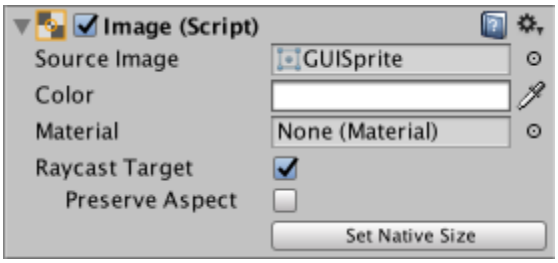
1. Text



The Text component, which is also known as a Label, has a Text area for entering the text that will be displayed. It is possible to set the font, font style, font size and whether or not the text has rich text capability.

There are options to set the alignment of the text, settings for horizontal and vertical overflow which control what happens if the text is larger than the width or height of the rectangle, and a Best Fit option that makes the text resize to fit the available space.

2. Image



An Image has a Rect Transform component and an Image component. A sprite can be applied to the Image component under the Target Graphic field, and its colour can be set in the Color field. A material can also be applied to the Image component. The Image Type field defines how the applied sprite will appear; the options are:

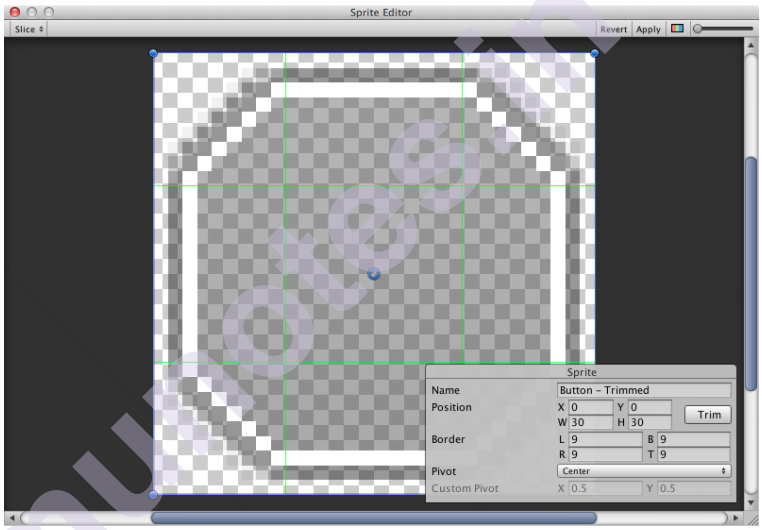
Simple - Scales the whole sprite equally.

Sliced - Utilises the 3x3 sprite division so that resizing does not distort corners and only the center part is stretched.

Tiled - Similar to Sliced, but tiles (repeats) the center part rather than stretching it. For sprites with no borders at all, the entire sprite is tiled.

Filled - Shows the sprite in the same way as Simple does except that it fills in the sprite from an origin in a defined direction, method and amount.

Images can be imported as UI sprites by selecting Sprite(2D / UI) from the 'Texture Type' settings. Sprites have extra import settings compared to the old GUI sprites, the biggest difference is the addition of the sprite editor. The sprite editor provides the option of 9-slicing the image, this splits the image into 9 areas so that if the sprite is resized the corners are not stretched or distorted.



3. Raw Image

The Image component takes a sprite but Raw Image takes a texture (no borders etc). Raw Image should only be used if necessary otherwise Image will be suitable in the majority of cases.

4. Mask

A Mask is not a visible UI control but rather a way to modify the appearance of a control's child elements. The mask restricts (ie, "masks") the child elements to the shape of the parent. So, if the child is larger than the parent then only the part of the child that fits within the parent will be visible.

5. Effects

Visual components can also have various simple effects applied, such as a simple drop shadow or outline.

Interaction Components

This section covers components in the UI system that handles interaction, such as mouse or touch events and interaction using a keyboard or controller.

The interaction components are not visible on their own, and must be combined with one or more visual components in order to work correctly.

1. Button

A Button has an OnClick UnityEvent to define what it will do when clicked.

2. Toggle

A Toggle has an Is On checkbox that determines whether the Toggle is currently on or off. This value is flipped when the user clicks the Toggle, and a visual checkmark can be turned on or off accordingly. It also has an OnValueChanged UnityEvent to define what it will do when the value is changed.

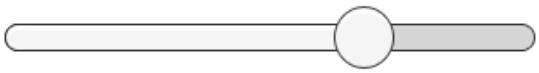
3. Toggle Group

A Toggle Group can be used to group a set of Toggles that are mutually exclusive. Toggles that belong to the same group are constrained so that only one of them can be selected at a time - selecting one of them automatically deselects all the others.

4. Slider

A Slider has a decimal number Value that the user can drag between a minimum and maximum value. It can be either horizontal or

vertical. It also has a OnValueChanged UnityEvent to define what it will do when the value is changed.



5. Scrollbar

A Scrollbar has a decimal number Value between 0 and 1. When the user drags the scrollbar, the value changes accordingly.

Scrollbars are often used together with a Scroll Rect and a Mask to create a scroll view. The Scrollbar has a Size value between 0 and 1 that determines how big the handle is as a fraction of the entire scrollbar length. This is often controlled from another component to indicate how big a proportion of the content in a scroll view is visible. The Scroll Rect component can automatically do this.

The Scrollbar can be either horizontal or vertical. It also has a OnValueChanged UnityEvent to define what it will do when the value is changed.

6. Dropdown

A Dropdown has a list of options to choose from. A text string and optionally an image can be specified for each option, and can be set either in the Inspector or dynamically from code. It has a OnValueChanged UnityEvent to define what it will do when the currently chosen option is changed.

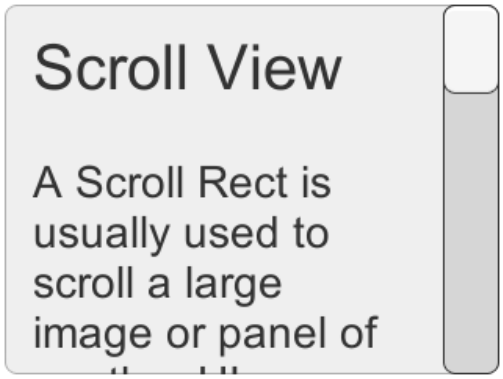
7. Input Field

An Input Field is used to make the text of a Text Element editable by the user. It has a UnityEvent to define what it will do when the text content is changed, and an another to define what it will do when the user has finished editing it.

8. Scroll Rect (Scroll View)

A Scroll Rect can be used when content that takes up a lot of space needs to be displayed in a small area. The Scroll Rect provides functionality to scroll over this content.

Usually a Scroll Rect is combined with a Mask in order to create a scroll view, where only the scrollable content inside the Scroll Rect is visible. It can also additionally be combined with one or two Scrollbars that can be dragged to scroll horizontally or vertically.



10.6 PUBLISHING BUILDS

The Build Settings window contains all the settings and options you need to publish your build to a variety of platforms. From this window you can create a **Development Build** to test your application, as well as publishing a final build. To adjust the publishing settings for your application's build go to **File** > **Build Settings**.

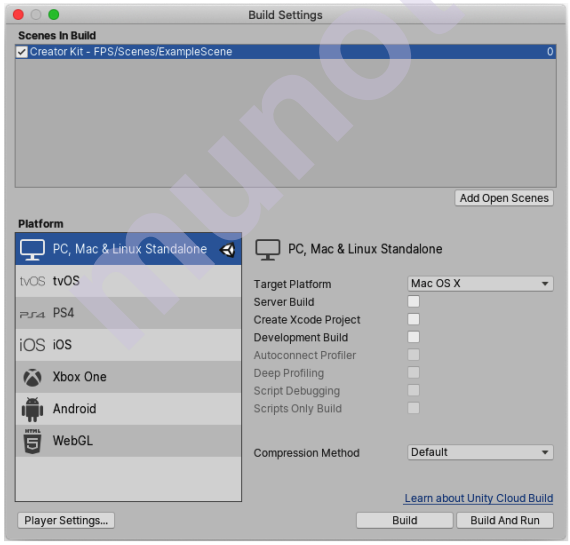


Fig: 10.5The Build Settings window

- Use the Scenes in Build panel to manage which Scenes Unity includes in the build. You can use the Platform section of the window to select which platform you want to build to, and adjust specific settings such as the Compression Method. These options vary depending on the Platform you select. For more information, see the documentation on Build Settings

- Select the Build or Build and Run button to begin the build process.

- You can choose a name and save location for your application through the Save dialog that appears. Note: depending on the platform you build to, Unity might only prompt you to choose a folder.
- When you select the Save button, Unity builds your application. If you are unsure where to save your build, consider making a subfolder inside your root folder to hold your builds.
- You cannot save the build into the Assets folder.

10.7 SUMMARY:

- Scripting is the process of writing blocks of code that are attached like components to GameObjects in the scene.
- MonoBehaviour is the base class from which every Unity script derives.
- A Network Manager, A user interface (for players to find and join games), Networked Player Prefabs (for players to control), Scripts and GameObjects are required for setting up multiplayer project.
- Unity provides three UI systems viz. UI Toolkit,The Unity UI package (uGUI),IMGUI.

10.8 QUESTIONS

1) State the difference between update(), FixedUpdate() and LateUpdate() method in Unity script.
2) Explain navigation and path finding in unity engine.
3) Write a note on setting up multiplayer project in unity.
4) Explain the concept of scripting in unity.
5) Explain the following Visual Components:
   a) Text
   b) Image
   c) Raw Image
   d) Mask
   e) Effects

10.9 REFERENCES

https://docs.unity3d.com/

❄❄❄❄❄❄❄