Academy of Interactive Entertainment

Advanced Diploma of Professional Games Development (Programming)

Lab – Maths Library

Semester No. 1 Year 1

It is important to understand the mathematical tools used in Three Dimensional Computer Graphics (3D Graphics). A 3D graphics program builds a scene composed of three-dimensional objects in three-dimensional space. Then two-dimensional images are produced from the 3D scene.

The scene is represented as a data structure in computer memory.
The 2D image is the 3D scene cut down to only the parts that are visible from the designated viewpoint, just like a photo is a 2D image of a 3D world.

3D Game models (mesh), model textures (2D image) and sprites (2D image) are stored as arrays of points and other data that tells the computer position, colour and lighting information for each pixel it is to display on the screen. Objects such as models and sprites or anything that is rendered (drawn) to the screen are collectively referred to as scene objects.

Vectors, matrices and quaternion are used to rotate, scale and translate (move) scene objects in the 3D scene. Each time the game loop runs through, the computer displays a 2D snapshot of that scene. Because this happens at more than 30 times a second, to our eyes it looks like a smooth and continual movement.

They are also used to perform calculations in the rendering pipeline (the process of taking the 3D scene and converting it to a 2D image) to display the correct information on the screen.

Even though we are creating 2D games, we will be using a 3D Renderer to display them, but to make it useable as a 2D Engine we will be ignoring the 3rd dimension (Z) as this is the depth into the screen.

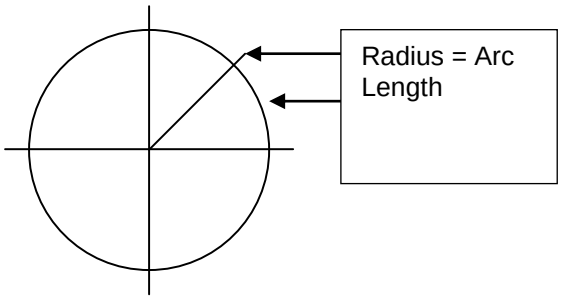
Now that you know some of the uses for math in games programming lets move on to what the terms are:

- Radians
- Scalars
- Primitives
- Coordinate systems
- Vectors
- Matrices

If you are interested in the details of mathematic and would like to know more than what is in this paper there is a web site http://skal.planet-d.net/demo/matrixfaq.htm

Radians

A radian is a measurement of an angle just, like a degree. One radian is the angle formed in any circle where the length of the arc defined by the angle and the radius are of the same length.



Radians are important because it is the unit that C++ math functions use for angles. Because you are probably accustomed to using degrees as your unit of measurement, you need to be able to convert from radians to degrees and vice – versa.
Pi (π) radians is the angle that contains half a circle. 180 degrees is also the angle that contains half a circle. We can use this information to convert both ways.

Degrees = Radians * 180 / π

Radians = Degrees * π / 180

An easy way to think about it is Pi (3.14) is the same as 180 degrees.
Therefore 2 * Pi (6.28) is 360 degrees
Pi / 2 (1.57) is 90 degrees and so forth.

I recommend in the maths library you will write to make yourself a couple of Deg2Rad and Rad2Deg functions if you find using Radians difficult.

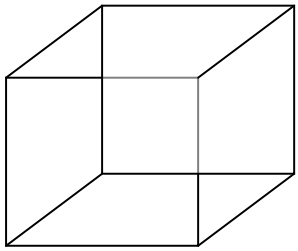
Scalars

Scalar is a universal descriptor for a single dimensional number, be it a float, int or any other real or complex number.
E.g. 2, 5.7

Primitives

In this section:
Points
Lines
Polygons

In both 2D and 3D, when you draw an object, you actually compose it with several smaller shapes called primitives. Primitives are 1 or 2 D entities or surfaces such as points, lines or polygons (a flat multisided shape) that are assembled in 3D space to form 3D objects. For example, a 3 dimensional cube consists of 6 2D squares, each placed on a separate face.



Points

A geometrical point is representation of a single coordinate in 3D or 2D space. In Graphics programming these points are called vertices. Each corner of a primitive is a vertex and, as you can see from a cube, those vertices are sometimes shared between primitives. The above cube has 8 vertices, yet each square has 4 vertices.

In games programming you will come across both points and vertices. They are called points when someone is referring to a location in space, but if someone is talking about the rendering (drawing on the screen) of an object, then it is referred to as a vertex.

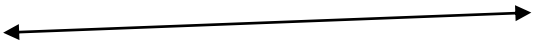
Lines

Line and line segments are made up of points.

A "line" is an infinite number of points that all lie on the same straight path. In game programming a line is, generally, defined with a starting point and a second point that the line passes through. So, in the following line, the first point defines where the line starts and the second point lies anywhere along that line.

A "line segment" is a straight path with two definite end points. In game programming a line segment is defined with a start point and an end point. The length of the segment is defined as the distance between those points.

Each side of the cube is made up of a line segment. Line segments can also be shared between primitives. There are 12 line segments in a cube yet each square has 4 line segments.



Polygons

Connecting three or more points in 3D space with line segments makes a polygon, unless the points all lie on the same line. Polygons in space are very important in 3D graphics as all objects that make up scenes are constructed out of triangles and other flat polygons.

Each square is a polygon so the cube is made up of 6 polygons.

Planes

A plane is a 2D cross section of 3D space. They stretch infinitely in two dimensions. Imagine one face of the square an infinite size, no edges defined. Planes are commonly defined in 1 of 3 ways:

By 3 Points each of which lie within the plane.
By 1 Point and a direction to which the plane is perpendicular to.
By a Direction and a distance from the origin

Planes are used in games programming to split up the scene into smaller areas for more efficient testing of visibility and collision.

Coordinate systems

This section:
Global Coordinates
Local Coordinates

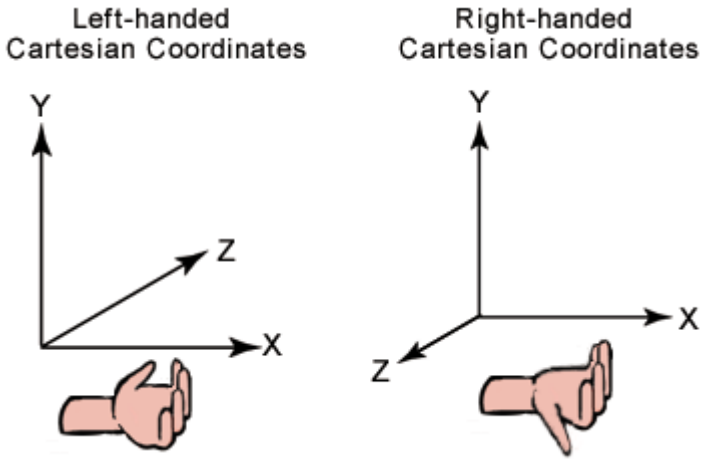
You can think about the points, lines, and polygon figures that compose the 3D scene without using a coordinate system. But in computer graphics we need to represent these objects, somehow, and to manipulate them. To do this, we use a coordinate system to give each object a unique set of coordinates that defines its position in the scene in relation to the origin.

The **origin** is the center of the scene where x = 0, y = 0 and z = 0.

The positive direction of the X, Y and Z axis varies from API to API so it is important to learn which way the coordinate system is oriented for each one you use. If the Physics Engine has its X axis pointing to the right and the Rendering engine has it pointing to the left you are in for a lot of fun unless you make allowances.

The Cartesian coordinate systems are based on a system where the axis are at right angles to each other (orthogonal).

Typically 3-D graphics applications use two types of Cartesian coordinate systems: left-handed and right-handed. In both coordinate systems, the positive x-axis points to the right, and the positive y-axis points up. You can remember which direction the positive z-axis points by pointing the fingers of either your left or right hand in the positive x-direction and curling them into the positive y-direction. The direction your thumb points, either toward or away from you, is the direction that the positive z-axis points for that coordinate system. The following illustration shows these two coordinate systems.



Although left-handed and right-handed coordinates are the most common systems, there is a variety of other coordinate systems used in 3-D software. For example, it is not unusual for 3-D modeling applications to use a coordinate system in which the y-axis points toward or away from the viewer, and the z-axis points up. In this case, right-handedness is defined as any positive axis (x, y, or z) pointing toward the viewer. Left-handedness is defined as any positive axis (x, y, or z) pointing away from the viewer.

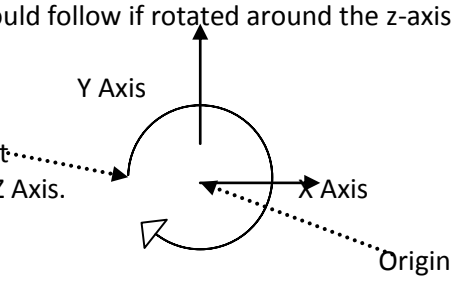
Exporters from modeling applications are written specifically for graphics engines to handle the adjustments required to change coordinate systems.

For our purposes, the X and Z axes will form a plane parallel to the ground, and the Y axis will define a line perpendicular to the ground. So a player walking through a playing field would be considered to be at some position in the XZ plane, and the distance between his head and his feet would be defined in terms of Y. This is by no means the only way to position a game element in a 3D coordinate system, but it is common, and this is the convention we will use.

Global Coordinates

Global coordinates are the coordinate system used by the 3D scene or world. The origin in the center of the scene and all objects are positioned in relation to and rotate around that origin.

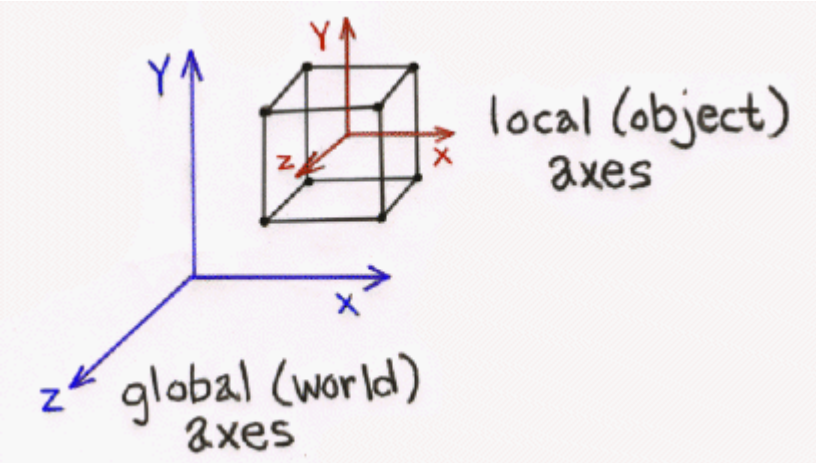
So if an object was placed at the global origin and you rotated it around the global axes, the object would spin on the spot. If the object was not at the origin, and you rotated it around the global axes, the object would move in a circle around the origin as shown below. The diagram shows a starting position and the path the object would follow if rotated around the z-axis which is not visible, as it points directly into the page.



Global coordinates are handy for setting up the scene and keeping track of where all players are but if you want to rotate a player on the spot, and they are not at the origin, it is not very useful.

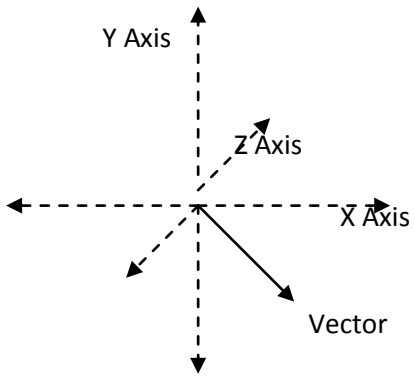
Local Coordinates

Any object that needs to move within the world (objects, players, etc) needs be defined within its own local coordinate system. A local coordinate system means that object's origin at its own center. There fore if you rotate a scene object around its local axis it spins on the spot, no matter where in the global coordinates it is placed.



Vectors

A vector is a geometrical object that has two properties: length and direction.
A vector's x, y and z values are the distance from the origin along each axis.



The vector above is about 7 units along positive x-axis, about 7 units along negative y-axis and 0 units along z-axis. We would represent this as (7, -7, 0) or as [7 –7 0].

Vectors are used for many purposes in 3D computer graphics.

Light from the sun (disregarding colour information) can be represented by a vector. The length of the vector is proportional to the brightness of the light; the longer the vector the brighter the light. Wind, also, is represented as a vector. Wind has direction and speed. A particle of dust travels in the direction of the wind. The length of the vector is proportional to the speed of the dust particle; the faster the particle the longer the vector

These are just a couple of examples.

The number of axis a vector has is called its dimension. A vector can contain as many dimensions as you like but in 3d graphics it usually has 2, 3 or 4.

Addition and subtraction

Vectors can be added or subtracted to form new vectors. This is done component by component.

Adding vector (A) to vector (B) to get vector (V)

$Vx = Ax + Bx$
$Vy = Ay + By$
$Vz = Az + Bz$

Subtraction is exactly the same except substitute a minus sign for the plus sign.

Vector addition can be done in any order but vector subtraction produces different results if done in different orders.

Scalar multiplication and division

Vectors can be scaled (made longer or shorter) by multiplying or dividing them by scalars. To do this, you multiply or divide each vector component by the scalar.

Multiply vector (A) by scalar (S) to get vector (V)

Vx = Ax * S
Vy = Ay * S
Vz = Az * S

Division is exactly the same except substitute a divisor sign for the multiplication sign.

Length

Using the Pythagorean theorem for right angle triangles, the length squared of the hypotenuse is equal to the sum of the square of each side; we can work out the length of any vector. The length is also called the magnitude of the vector.

This is how the magnitude of vector v is represented
||v||

To get length (L) of Vector (V) in 3D

$L = \sqrt{Vx^2 + Vy^2 + Vz^2}$

Normalization

Some vector calculations will require a vector of length 1.0. This is usually when just a direction is wanted, not the length.

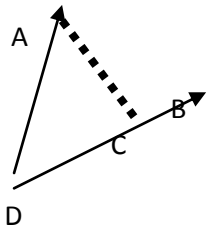
To force a vector to have a length 1.0, you must normalize the vector, or in other words divide the components of the vector by its length.

To get the normal vector(N) of vector (V)

Nx = Vx / ||V||
Ny = Vy / ||V||
Nz = Vz / ||V||

Dot product

The dot product of two vectors, also known as the scalar product or inner product, is one of the most heavily used operations in 3D graphics, physics and collision because it supplies a measure of the difference between the directions in which the two vectors point. The dot product is a scalar that represents the number of units a vector projects onto another vector.
In the example below there are two vectors A and B and the dotted line shows the projection of A onto B. The resultant vector of the projection is DC and the dot product is the length of DC.



The dot product is given by the sum of the products of each component.

A . B = AxBx + AyBy + AzBz

This yields a scalar result that tells us how close the two vectors are to pointing in the same direction. A result of 0 means the two vectors are orthogonal (perpendicular). A zero vector (v = (0, 0, 0)) is orthogonal to every vector since v . A = 0. Any vector on the same side of the plane that is perpendicular to A will yield a positive dot product and any vector that is on the opposite side of the plane will yield a negative dot product.

This site has a great demo to play with to get the idea
http://www.falstad.com/dotproduct/

If both vectors are normalised (have a unit length of 1) if they are both facing in the same direction the result from the dot product will be 1. If they are at right angles to each other the result will be 0. If they are opposite facing the result will be -1.

The dot product can be used to determine the angle between 2 normalised vectors, the result from the dot produce is used with arccosine and will give you the angle between them in radians.

Matrices

Matrices in computer graphics are used for transforming vectors and points in the global and screen space. Even though a matrix requires more processing to set up, its efficiency with calculating translation, scaling, shearing and rotation per vertex out performs any other process.

A matrix is a table of numbers and is stored as an array when it comes to programming.
The order of a matrix is the size of it so a matrix with n rows and m columns is said to have order n *m.
The most commonly used matrices for 3D programming are 3 x 3 and 4 x 4.

A 4x4 matrix is displayed as:

$$M = \begin{bmatrix} A & E & I & M \\ B & F & J & N \\ C & G & K & O \\ D & H & L & P \end{bmatrix}$$

There are three directional vectors contained in a 4x4 matrix and they define the coordinates for the X, Y and Z axis.

The vectors run down the columns. So from the matrix above, the direction vector for each axis is as follows:

$$\text{X-axis} = \begin{bmatrix} A \\ B \\ C \end{bmatrix} \qquad \text{Y-axis} = \begin{bmatrix} E \\ F \\ G \end{bmatrix} \qquad \text{Z-axis} = \begin{bmatrix} I \\ J \\ K \end{bmatrix}$$

The fourth value of each directional vector [D H L] is a value used for transforming the global coordinates to the screen coordinates and is usually set to 1 initially. This is referred to as the w value of a vector.
The fourth vector [M N O P] alters the perspective, which is adjusting the size of an object according to the distance it is from the origin along the z axis (or whichever axis moves into the screen), and is usually set to (0, 0, 0, 1). It causes a flat 2d image to appear as an image with depth.

OpenGL matrices store their vectors in matrices as shown above, however the way these are stored vary between APIs. Direct3D stores its vectors across the matrix rows (known as row-major) instead of down the columns. Renderware Graphics sets theirs up completely different again – the vectors are read along the rows but the vectors are not the axis directional vectors. It is a good idea to check how each API treats matrices before using them in your code.

Types of matrices

There are different types of matrices that are used in graphics programming.

A **scaling** matrix is used for scaling geometry in the x, y and z directions.

$$\begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

(Where S is the scaling factor and can be different values)

You should always try to avoid using scaling matrices in games.

A **rotation** matrix is used for rotating geometry.

This is the rotation matrix for rotation around the **z axis**:

$$\begin{bmatrix} \cos(V) & \sin(V) & 0 \\ -\sin(V) & \cos(V) & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(Where V is the angle to rotate in radians)

Rotation for 2D is a 2 x 2 matrix with the same cosine and sine values as the matrix above.
Simply remove the 3rd row and 3rd column.

This is the rotation matrix for rotation around the **y axis**:

$$\begin{bmatrix} \cos(V) & 0 & -\sin(V) \\ 0 & 1 & 0 \\ \sin(V) & 0 & \cos(V) \end{bmatrix}$$

And for the **x axis**:

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(V) & \sin(V) \\ 0 & -\sin(V) & \cos(V) \end{bmatrix}$$

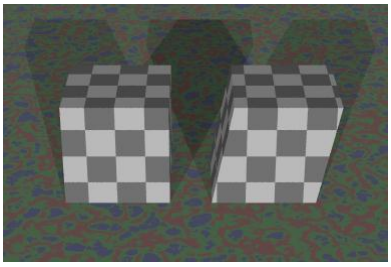
A **translation** matrix is used for moving geometry:

$$\begin{matrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{matrix}$$

Where dx, dy, dz is the position we will translate by.

There are also:

- **Zero** matrices where all values are zero.
- **Identity** matrices that are the same as a scaling matrix but all values of S are 1.
- **Shearing** matrices that distort the shape of an object in a non-orthogonal way.
  The easiest shear matrix to construct works by shifting the object by a certain amount for every unit amount moved along one of the axes. It is hard to describe so here is an image of an object that was sheared a small amount in the x direction for every unit it extends in the y direction.



The right object is the left object after a shearing matrix has been applied.

Getting Z Rotation from a Matrix

You will want to get the Rotation from a Matrix from time to time, this can be done using arctangent of the Y Vector, and checking the direction of the X Vector to see if it is pointing left or right.

Another better way is by using the atan2 function in cmath that does this for you, and effectively returns an angle from -180 to +180 degrees (actually -PI to +PI Radians)

Addition and subtraction

Matrix addition and subtraction are done exactly the same way as the vector addition and subtraction. Each element of two matrices are added, or subtracted, to form a third matrix.

$$\begin{bmatrix} A & C \\ B & D \end{bmatrix} + \begin{bmatrix} 1 & 3 \\ 2 & 4 \end{bmatrix} = \begin{bmatrix} A+1 & C+3 \\ B+2 & D+4 \end{bmatrix}$$

Multiplication and division

To multiply or divide a matrix by a scalar, you multiply or divide each matrix element by a scalar.

$$\begin{bmatrix} A & C \\ B & D \end{bmatrix} * \text{scalar} = \begin{bmatrix} A * \text{scalar} & C * \text{scalar} \\ B * \text{scalar} & D * \text{scalar} \end{bmatrix}$$

To multiply two matrices together the first matrix must have the same number of columns as the second matrix has rows.

Multiply each element in the first row of the first matrix with each element of the first column of the second matrix and add the result together.
Then multiply each element in the first row of the first matrix with each element second column of the second matrix (if any) and add the result together.
Repeat this with each column of the second matrix.

Repeat the above using the second row of the first matrix with each column of the second matrix.
Repeat for each row of the first matrix.

Multiplying two 3 * 3 matrices together

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} * \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix} = \begin{bmatrix} (A*1+D*2+G*3) & (A*4+D*5+G*6) & (A*7+D*8+G*9) \\ (B*1+E*2+H*3) & (B*4+E*5+H*6) & (B*7+E*8+H*9) \\ (C*1+F*2+I*3) & (C*4+F*5+I*6) & (C*7+F*8+I*9) \end{bmatrix}$$

It is important to remember that swapping around the order of multiplication of matrices doesn't produce the same result. When using matrices to rotate and transform points, vectors or other matrices, you need to think about the order in which things should happen. Look at it like you are concatenating matrices together. They happen in the order that you multiply them in.

There is an order of rotation for rotating around more than one axis. The order is:
First rotate around the x- axis, then the y-axis and finally the z-axis. So you would multiply the rotation matrices together in that order to get the result you would be expecting.

Points and vectors can be represented by column matrices and then multiplied by other matrices. This is how a point or vector can be translated, rotated and transformed in other ways.

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} x * A + y * D + z * G \\ x * B + y * E + z * H \\ x * C + y * F + z * I \end{bmatrix}$$

Homogenous matrices are a little different, we will be using a 3x3 matrix for 2 dimensions. This allows us to store translation. As they store translation, they can be used to translate vectors and points. General only points want to be translated through space to their new location; vectors being a direction do not need to be translated as this will totally change their values.

Transform a 2D vector

This transformation will only take rotation and scale information from the matrix, and apply it to the vector.
This is useful if you wish to only rotate and scale direction vector.
How to transform a 2D vector by a homogenous matrix:

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x * A + y * D \\ x * B + y * E \end{bmatrix}$$

Transform and translate a point

Notice how we are just multiplying the vector by A,B,D,E. These are the values within the 2D matrix that stores the rotation and scale. G H store the translation, thus to transform and translate a point we need the following:

$$\begin{bmatrix} A & D & G \\ B & E & H \\ C & F & I \end{bmatrix} * \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x * A + y * D + G \\ x * B + y * E + H \end{bmatrix}$$

We are just simply adding the translation of G, H onto the respective components of the vector.