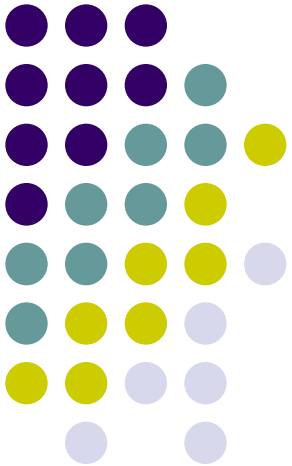
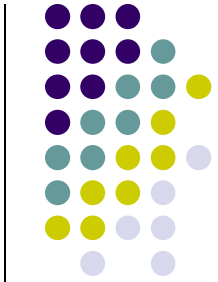
Advanced Game Engine Creation

Lecture 2

Maths for 3D Worlds



Objectives



Last week we:

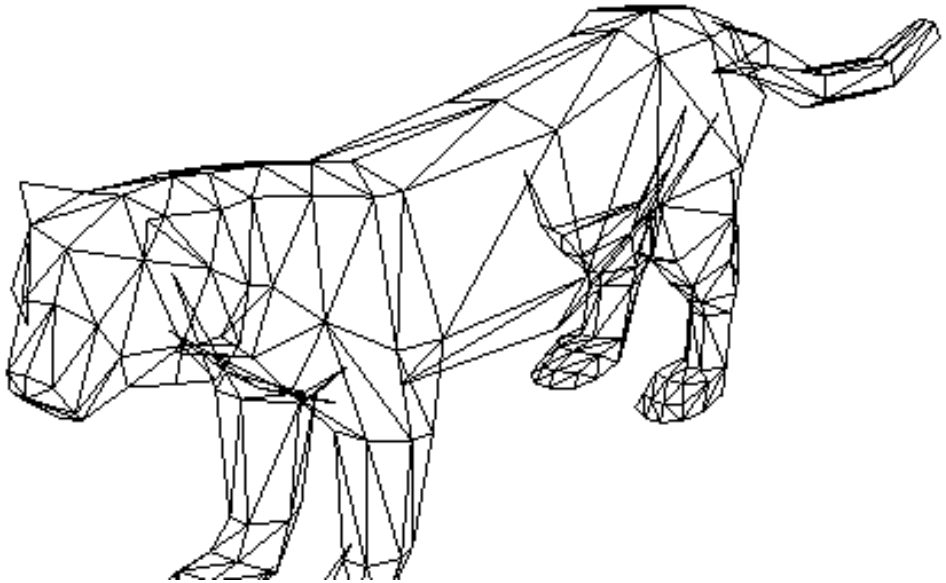
- introduced the module
- quickly reviewed C++
- introduced OpenGL and its incorporation into the SDL framework
- using OpenGL functions to display triangles and 3D shapes

In this lecture, we will

- review 3D points and vectors
- look at vector operations – addition, subtraction, scalar multiplication
- discuss C++ operator overloading
- create a 3D vector struct with overloaded operators
- introduce matrices
  - multiplying vectors by matrices
  - multiplying matrices together

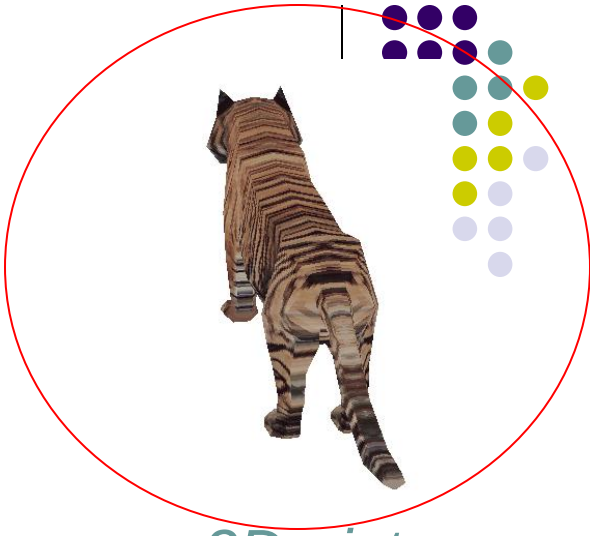
Why maths?

- game world is represented in 3D space by models made up of triangles made up of vertices

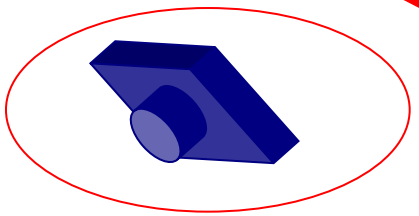


Every frame:

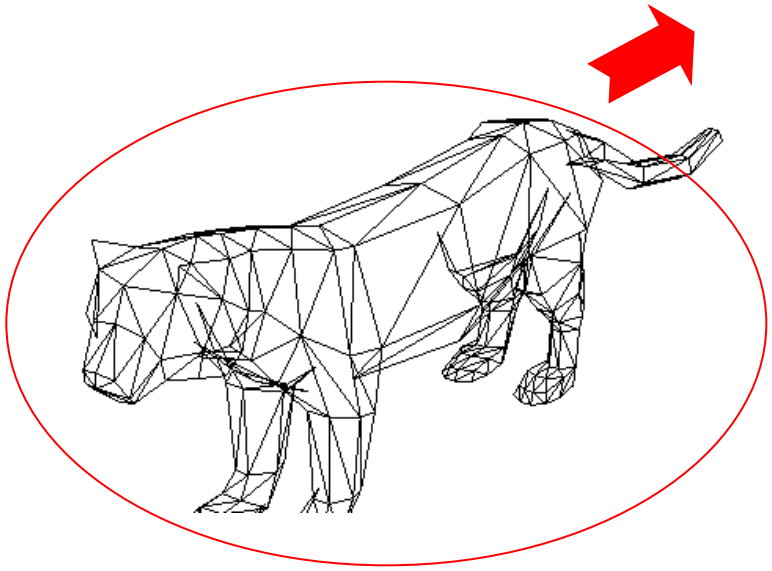
- 3D viewing is like **taking a picture with a camera**



2D picture



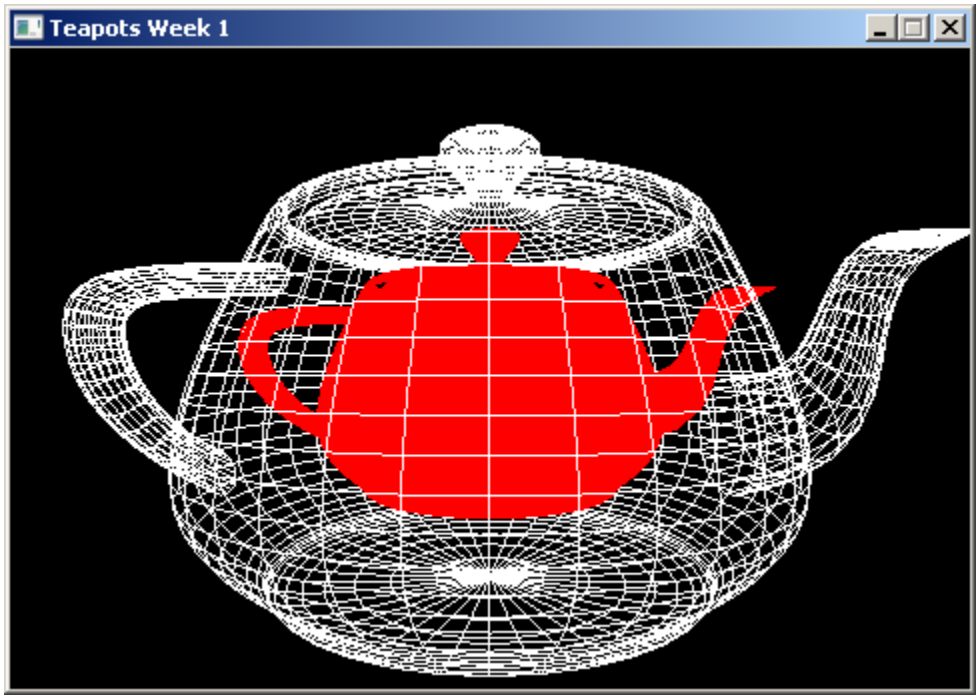
Camera



3D scene

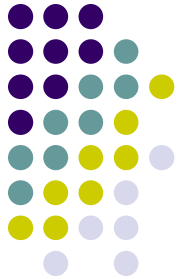
- use user input, physics, AI, animation to update position of game objects
- render view of 3D game world as a 2D picture

Last week – the teapot scene



- how can we:
  - display teapots in different positions?
  - rotate and scale them?
  - render the solid teapot so it doesn't look flat?

The secret of 3D rendering



- we keep only one copy of each model
  - Ideally centred on the origin
- each model is scaled, rotated and placed into position every frame
  - then rendered and discarded
  - multiple times to display multiple objects
- start afresh next frame

Right-handed 3D coordinates

- the positive z axis comes out of the screen/paper
- used by OpenGL

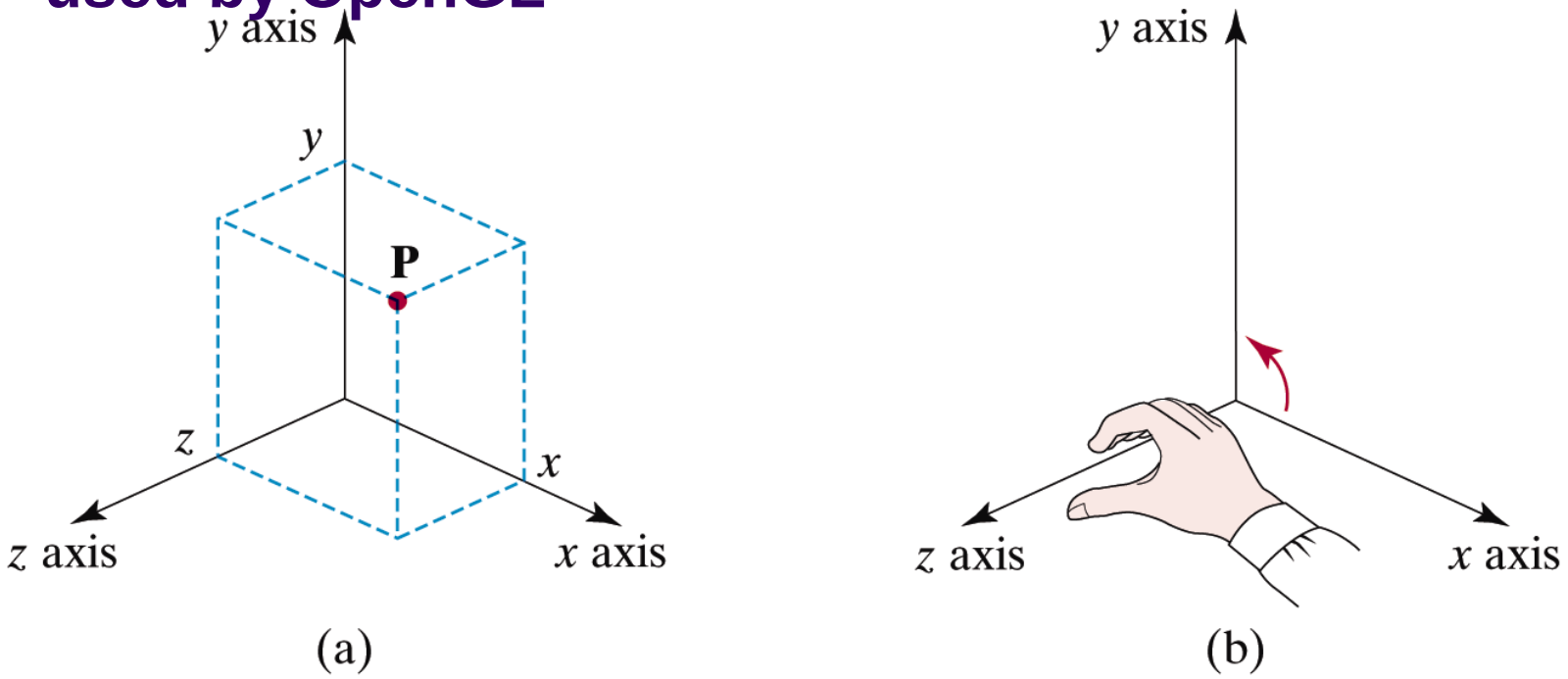
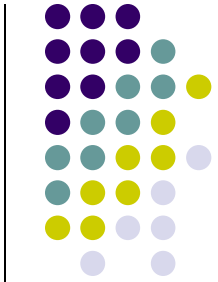


Figure A-6

Coordinate representation for a point **P** at position (x, y, z) in a standard right-handed Cartesian reference system.

Computer Graphics with Open GL, Third Edition, by Donald Hearn and M.Pauline Baker.
ISBN 0-13-0-15390-7

3D Space

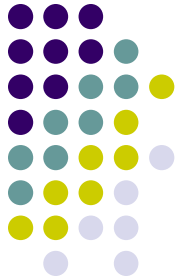


- We need two elements available in our 3D space

- Points
  - These are used to define positions
    - An object is located at a point
    - A vertex is located at a point

- Vectors
  - These are used to define directions
    - A viewer is looking in a certain direction
    - A projectile is moving in a certain direction

Representation of points

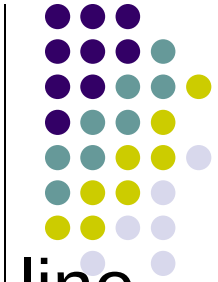


- the location of a point in 3D space can be represented by 3 floating point numbers
  - (x, y, z) where x, y and z are floats
  - (0.0, 0.0, 0.0) the origin
    - represented by (0.0f, 0.0f, 0.0f) in an OpenGL program
  - (3.2, 0.6, 4.5)

Vectors

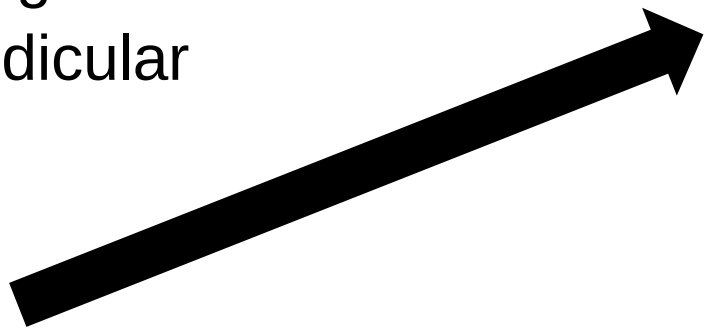
- A vector refers to a quantity that possesses both magnitude and direction
  - 45 mph north-east
  - 4 km/h towards the pub
- while a scalar just has magnitude
  - I have driven 18 miles
  - I'm 4 km from the pub
- don't confuse *vector* the mathematical construct with *vecto*r the C++ container

Vectors



- We graphically specify a vector by a directed line segment, where the length denotes its magnitude and the aim denotes its direction
- Note that where we draw a vector is immaterial as changing its location does not change its magnitude or direction
- we use *unit* vectors (a vector of length 1) if we are only interested in the direction
  - the direction an object is pointing at
  - the direction of a vector perpendicular

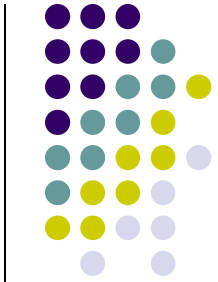
  to a surface (for lighting)



Vectors

- Computers cannot work with vectors geometrically, therefore we need to represent them numerically
- So, in our 3D space we
  - Define some point as the origin (0, 0, 0)
  - Assume basis vectors for this 3D space
  - Translate all vectors so that their tails are at this origin
    - This is called standard position
  - We can now represent a vector as the coordinates of its head $\mathbf{v} = (x, y, z)$
- Our vector is now defined relative to our 3D coordinate system with origin at (0, 0, 0)

Vectors and Points

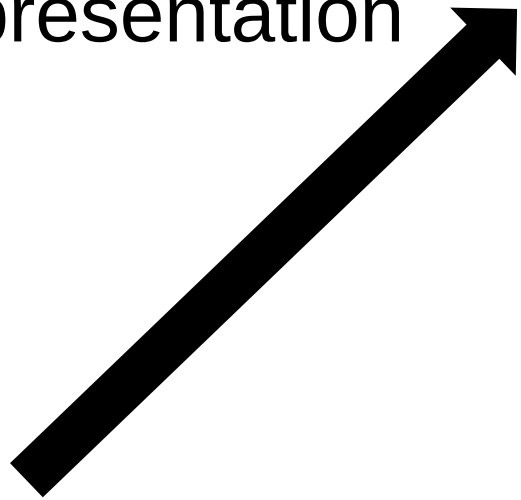


- Both vectors and points can be described by coordinates (x, y, z) relative to a frame.
- However, they are not the same
- A point represents a location in 3D-space, whereas a vector represents a magnitude and direction

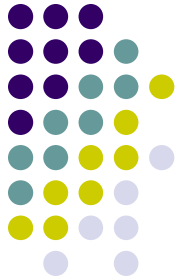
Vector magnitude

- the magnitude of a vector is simply its length
- can calculate from its [x,y,z] representation using Pythagoras theorem
- if vector $\boldsymbol{v} = (x, y, z)$
- then the magnitude of v is

  $\|\boldsymbol{v}\| = \sqrt{(x^2 + y^2 + z^2)}$
- Note that magnitude is a scalar value



Unit vectors

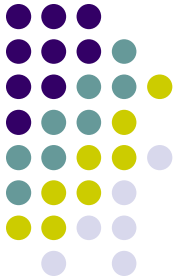


- a unit vector $\hat{v}$ has magnitude 1
  - denotes direction only
  - use when we don't care about the magnitude
    - direction a character is facing
- We can normalise a vector (calculate its unit vector) by dividing each of its components by its magnitude
- for vector v $= (x, y, z)$

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

- the unit vector is $\hat{v} = (x/||v||\ y/||v||,\ z/||v||)$

Worked example

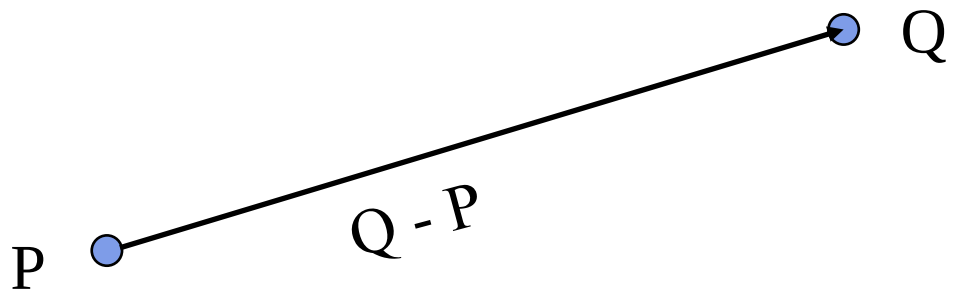


- calculate the magnitude of the vector (3, 5, 4)
- calculate the unit vector corresponding to the vector (3, 5, 4)

Vectors

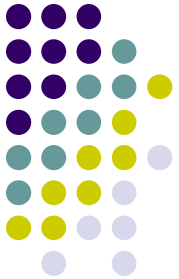
- Given points P and Q, we can easily find the vector that lies between them

- The vector is Q – P



- Note the direction of (Q – P) is from P to Q
- when P is the origin (0,0,0), the point Q has the same coordinates as the vector between P and Q

Worked example

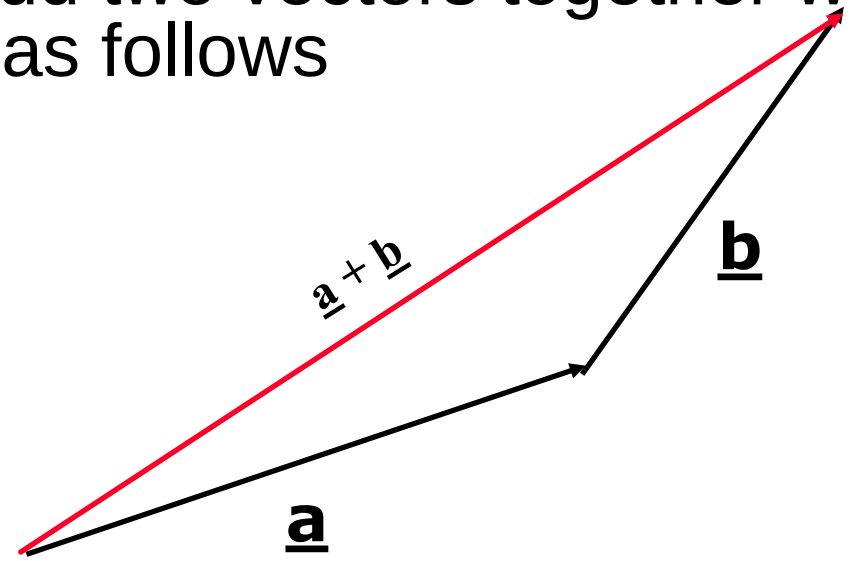


- calculate the vector PQ between the points

P = (3, 6, -2) and Q = (-4, 2, 0)

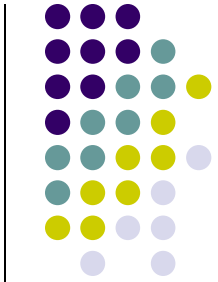
Vector addition

- If we add two vectors together we get another vector as follows



- Let a = (a_x, a_y, a_z) and b = (b_x, b_y, b_z)
- Then a + b= (a_x + b_x, a_y + b_y, a_z + b_z)
- multiple forces acting on an object
- a translation of a position a by a vector b

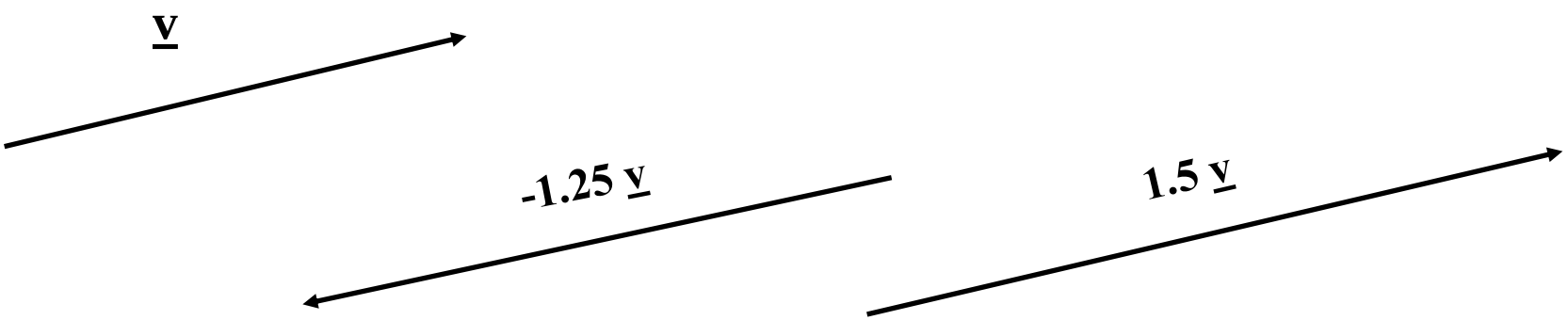
Worked example



- calculate the new position of a point (-4, 6, 0) translated by the vector (2, 3, 4)

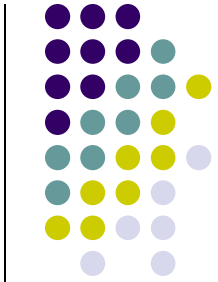
Vectors

- We can multiply vectors by scalars



- If we multiply a vector **v** by a scalar ***t*** we get a vector ***t*****v** which is
  - Parallel to **v**
  - ***t*** times the length of **v**
- Let *t* be a scalar, and let $v = (v, v_y, v_z)$
- Then $tv = (tv_x, tv_y, tv_z)$

Worked examples



- multiply the vector (-8, 3, 2) by the scalar 2.5
- multiply the vector (-8, 3, 2) by the scalar -1
  - what is the effect of this operation?

A Vector3D struct or class

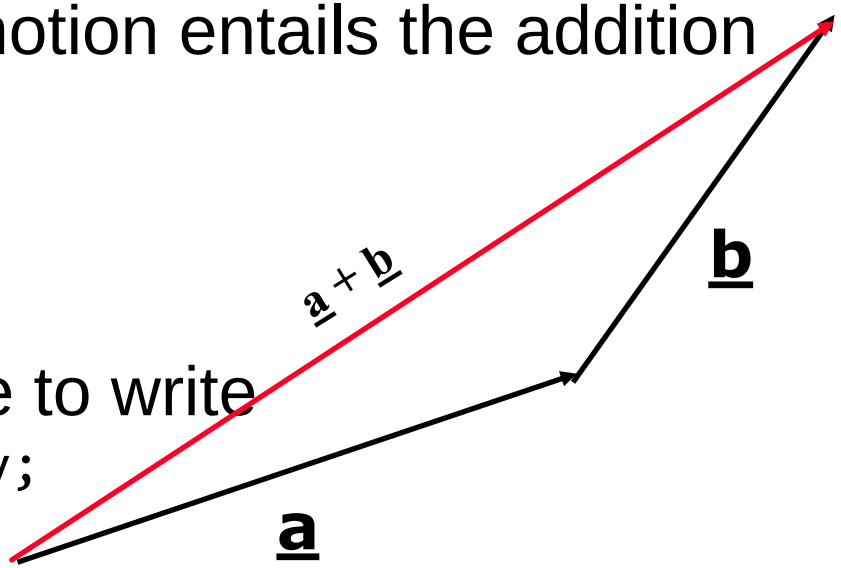
- its is a good idea to write a Vector3D struct or class which includes common vector operations
  - struct has public members by default, otherwise same as class
- for example, simple linear motion entails the addition of two vectors every frame

```
position.x += velocity.x;
position.y += velocity.y;
position.z += velocity.z;
```

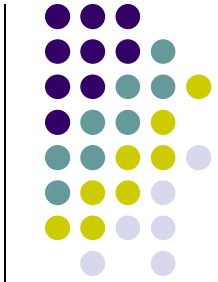
- it would be neater to be able to write

```
position = position + velocity;
```

- or even `position += velocity;`
- where position and velocity are both Vector3D instances



Overloading operators



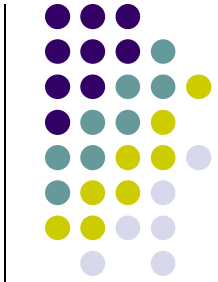
- we can overload (write new versions of) operators in C++
  - similar to overloading functions
  - most languages don't allow you to do this (why not?)

```
Vector3D  operator+( const Vector3D b) {
  Vector3D result;
  result.x = x + b.x;
  result.y = y + b.y;
  result.z = z + b.z;
  return result;
}
```

usage: `position = position + velocity;`

where `position` and `velocity` are both instances of `Vector3D`

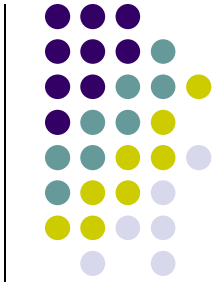
Overloading the += operator



```
Vector3D& operator += (const Vector3D b) {
  x = x + b.x;
  y = y + b.y;
  z = z + b.z;
  return *this;
}
```

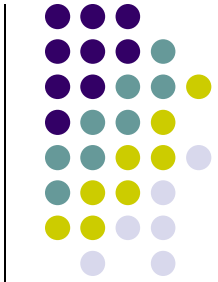
- usage position += velocity;

The need for matrices



- it is very useful to represent transformations using matrices
  - rotations about an axis, translations, scaling
- we can combine individual transformations by multiplying them together
- we can then use the final transformation matrix **M** to transform all the vertices of a game object
  - simply multiply each vertex by **M**
- we will cover transformations in more detail next week
- this week we will just look at the mechanics of matrix multiplication

Multiplying a vector by a matrix



- we shall use the **M.$\underline{\mathbf{v}}$** notation
- M is a matrix and $\underline{v}$ is a vector which represents the coordinates of a point
  - This means that the matrix M premultiplies the vector $\underline{v}$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax + by + cz \\ dx + ey + fz \\ gx + hy + iz \end{bmatrix}$$

Worked example

$$\begin{bmatrix} 1 & 0 & 3 \\ 0 & -2 & 1 \\ 2 & -1 & -1 \end{bmatrix} \begin{bmatrix} 2 \\ -1 \\ 3 \end{bmatrix} = \begin{bmatrix} 1*2+0*-1+3*3 \\ 0*2+-2*-1+1*3 \\ 2*2+-1*-1+-1*3 \end{bmatrix}$$

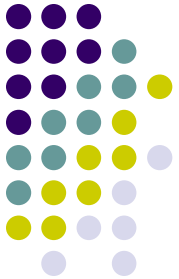
$$= \begin{bmatrix} 2+0+9 \\ 0+2+3 \\ 4+1-3 \end{bmatrix} = \begin{bmatrix} 11 \\ 5 \\ 2 \end{bmatrix}$$

Multiplying two matrices

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x & p & s \\ y & q & t \\ z & r & u \end{bmatrix} =$$

$$\begin{bmatrix} ax + by + cz & ap + bq + cr & as + bt + cu \\ dx + ey + fz & dp + eq + fr & ds + et + fu \\ gx + hy + iz & gp + hq + ir & gs + ht + iu \end{bmatrix}$$

Worked example – multiplying two matrices

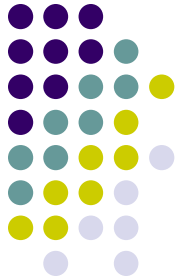


$$\begin{bmatrix} 1 & 2 & 3 \\ 0 & -1 & 2 \\ 0 & 2 & -2 \end{bmatrix} \begin{bmatrix} 2 & -1 & 4 \\ 3 & 1 & 0 \\ 1 & 0 & -1 \end{bmatrix}$$

$$= \begin{bmatrix} 2+6+3 & -1+2+0 & 4+0-3 \\ 0-3+2 & 0-1+0 & 0+0-2 \\ 0+6-2 & 0+2+0 & 0+0+2 \end{bmatrix}$$

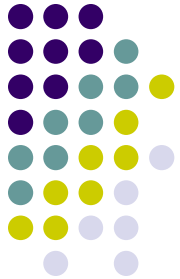
$$= \begin{bmatrix} 11 & 1 & 1 \\ -1 & -1 & -2 \\ 4 & 2 & 2 \end{bmatrix}$$

Some more useful 3D maths yet to cover



- vector dot and cross products
- calculating the direction of a vector from its [x, y, z] representation
- using matrices to represent transformations (next week)

Summary



Today we have

- reviewed 3D points and vectors
- looked at vector operations – addition, subtraction, scalar multiplication
- discussed C++ operator overloading
- created a 3D vector struct with overloaded operators
- introduced matrices
  - multiplying vectors by matrices
  - multiplying matrices together

Practical

- Pen and paper calculations
- implement and test a C++ Vector3 struct

Further work

- Google for vector and matrix calculators and animations!