MANCHESTER 1824

COMP27112

Computer Graphics and Image Processing

3: Transformations

Toby.Howard@manchester.ac.uk

1

MANCHESTER 1824

Introduction

We'll look at:

- Types of geometrical transformation
- Vector and matrix representations
- Homogeneous coordinates
- Using transformations in OpenGL
- Some handy vector geometry

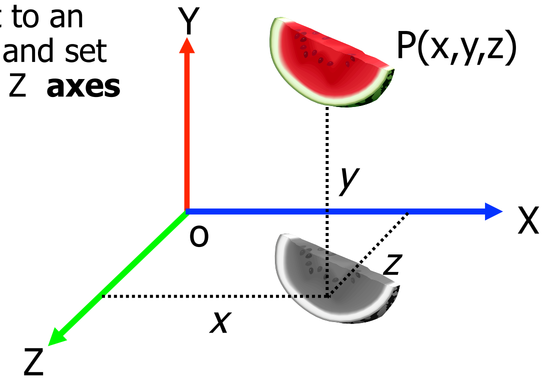
2

MANCHESTER 1824

3D Cartesian coordinates

René Descartes, 1596-1650, inventor of Cartesian coordinates

- A coordinate represents a point in space, measured with respect to an **origin** and set of X, Y, Z **axes**

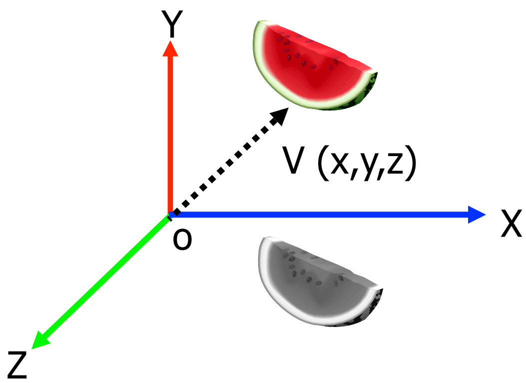


3

MANCHESTER 1824

Vectors

- A vector represents a **direction** in space, with respect to a set of X, Y, Z axes. It has a characteristic length. A vector of length 1 is known as a “unit vector”



4

MANCHESTER 1824

Coordinates and Vectors

- Both coordinates and vectors can be represented by a triple of x,y,z values. In computer graphics we usually write these in column format:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- In the previous slides
  - The melon is **at** Point **P**
  - The **spatial relationship** between the origin and the melon is described by the vector **V**

5

MANCHESTER 1824

Danger: 2 different representations

- We can write a vector as either a column or a row:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \text{or} \quad \begin{bmatrix} x & y & z \end{bmatrix}$$

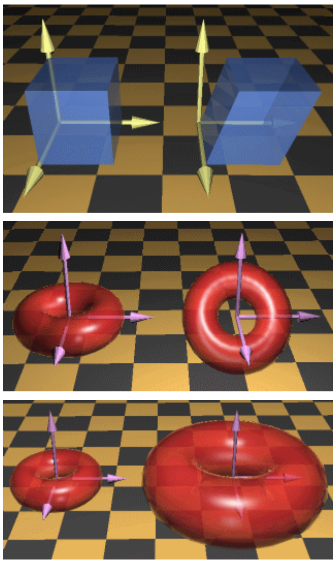
- OpenGL uses column vectors
- MATLAB uses row vectors
- Yes, this is confusing!
- The two representations are equivalent, but a transformation matrix used with column vectors is the **transpose** of the equivalent matrix used with row vectors

6

MANCHESTER 1824

Geometrical Transformations

- We define the shape of a geometric primitive using points
- We can apply geometrical transformations to points to change them
- These include translation, scaling and rotation
- To transform an entire shape, we transform all its individual points



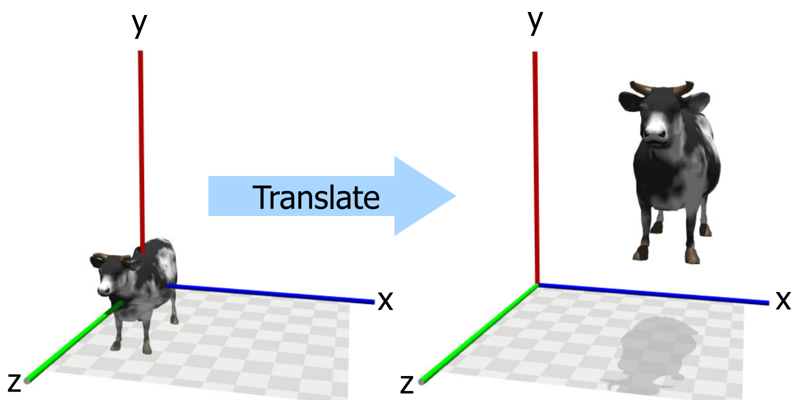
7

MANCHESTER 1824

Translation

$$x' = x + tx$$
$$y' = y + ty$$
$$z' = z + tz$$

- Applies a 3D shift (tx, ty, tz) to all coordinates

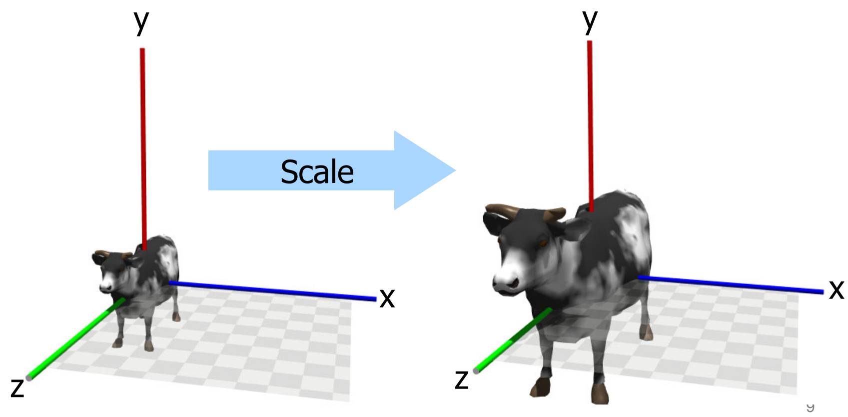


8

Scaling

- Applies a scale factor (sx, sy, sz) to all coordinates (with respect to the origin)

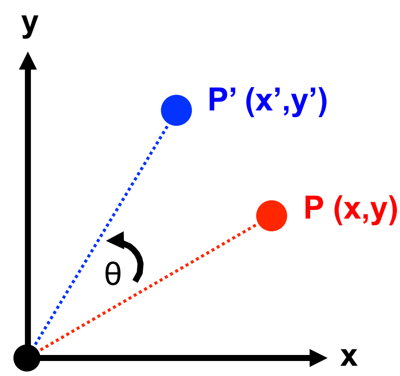
$$x' = x \,.\, sx$$
$$y' = y \,.\, sy$$
$$z' = z \,.\, sz$$



9

Rotation (2D)

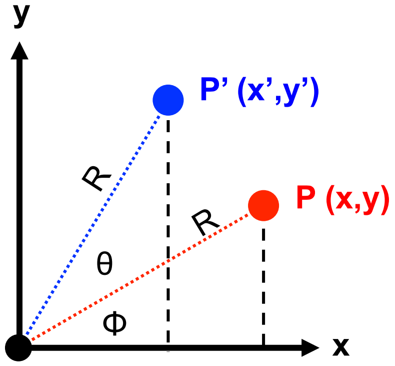
- Before we look at 3D, let's look at the 2D case.
- We want to rotate point **P** about the origin, by angle θ



10

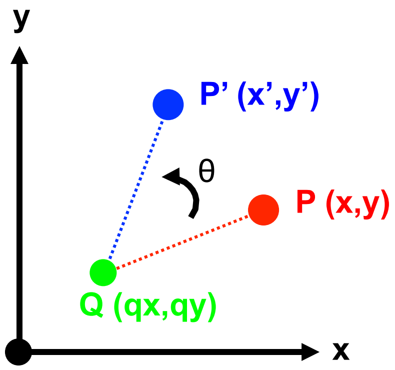
Rotation (2D)

- x= RcosΦ
- y= RsinΦ
- x′= Rcos(θ+Φ)
  - x′= RcosΦcosθ - RsinΦsinθ
- y′= Rsin(θ+Φ)
  - y′= RcosΦsinθ + RsinΦcosθ
- Substituting for RcosΦ and RsinΦ gives:
- x′= xcosθ – ysinθ
- y′= xsinθ + ycosθ



Rotation (2D) about a point

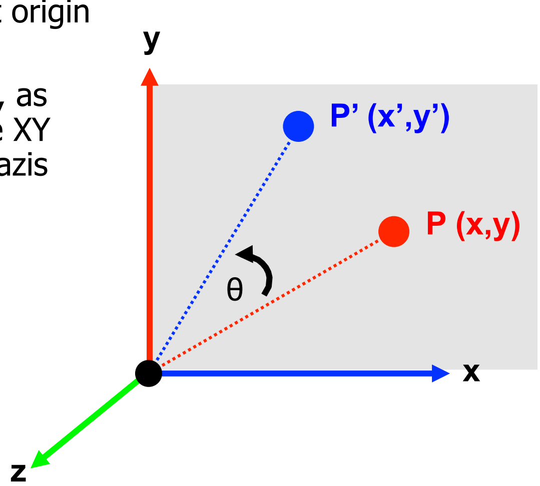
- What if we want to rotate about some other point than the origin?
- How do we rotate point **P** about point **Q** ?
- We break the problem into 3 simpler steps:
  - Translate by (-qx,-qy) to place Q at the origin
  - Do the rotation by θ
  - Translate back again by (+qx, +qy)
- This is **always the way** we do complex transformations… see later



MANCHESTER
1824

Rotation (3D)

- Rotation of **P** about origin in 2D…
- …is the same in 3D, as rotation of **P**, in the XY plane, about the Z-azis
- So:
- $x' = x\cos\theta - y\sin\theta$
- $y' = x\sin\theta + y\cos\theta$
- $z' = z$ (no change)

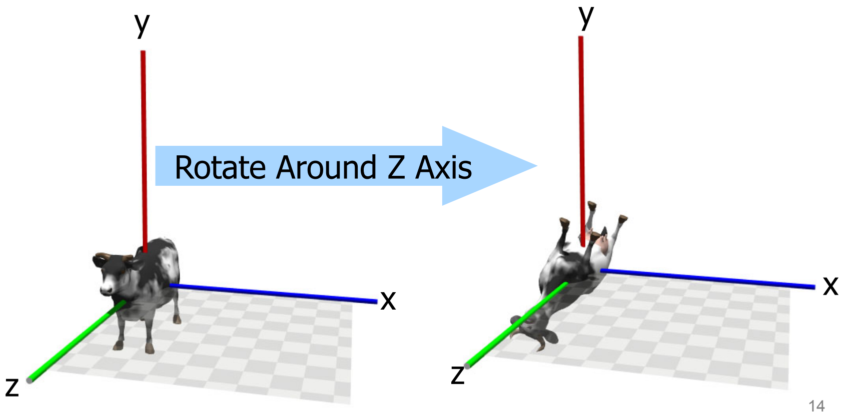


MANCHESTER
1824

Rotation (3D)

$$x' = x \,.\, \cos\theta \; - y \,.\, \sin\theta$$
$$y' = x \,.\, \sin\theta \; + y \,.\, \cos\theta$$
$$z' = z$$

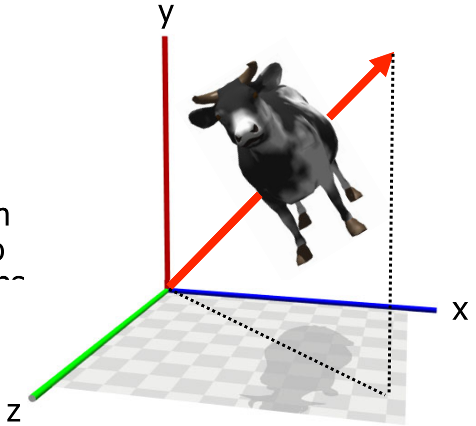
- Rotations are relative to an axis, e.g. a rotation by angle θ about the Z axis



MANCHESTER 1824

Rotation (3D) about a vector

- In 3D we often want to rotate (or scale) about an **arbitrary** axis vector
- This is analogous to the 2D case of rotating (or scaling) about an arbitrary point
- And we approach it in the same way: we do it as sequence of steps (which we will describe later)



15

MANCHESTER 1824

Representing transformations

- We've seen the following equations for representing some transformations:

Translation	Scale	Rotation (about Z)
$x' = x + tx$	$x' = x \,.\, sx$	$x' = x \,.\, \cos\theta \; - y \,.\, \sin\theta$
$y' = y + ty$	$y' = y \,.\, sy$	$y' = x \,.\, \sin\theta \; + y \,.\, \cos\theta$
$z' = z + tz$	$z' = z \,.\, sz$	$z' = z$

- They're all different!
- It would be very convenient if we could use a single **homogeneous** (== "the same") representation
- We use **vectors** and **matrices**

16

MANCHESTER 1824

Using matrices: scaling

- A transformation changes a vector into another vector
- We can represent this change using a **matrix**
- Example, scale (x,y,z) by (2,3,5):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Multiply elements row by column:
  - **x'** = 2*x + 0*y + 0*z = **2x**
  - **y'** = 0*x + 3*y + 0*z = **3y**
  - **z'** = 0*x + 0*y + 5*z = **5z**

17

MANCHESTER 1824

Using matrices: scaling

- A transformation changes a vector into another vector
- We can represent this change using a **matrix**
- Example, scale (x,y,z) by (2,3,5):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} \boxed{2 \quad 0 \quad 0} \\ 0 \quad 3 \quad 0 \\ 0 \quad 0 \quad 5 \end{bmatrix} \cdot \boxed{\begin{bmatrix} x \\ y \\ z \end{bmatrix}}$$

- Multiply elements row by column:
  - **x'** = 2*x + 0*y + 0*z = **2x**
  - **y'** = 0*x + 3*y + 0*z = **3y**
  - **z'** = 0*x + 0*y + 5*z = **5z**

18

Using matrices: scaling

- A transformation changes a vector into another vector
- We can represent this change using a **matrix**
- Example, scale (x,y,z) by (2,3,5):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Multiply elements row by column:
  - **x'** = 2*x + 0*y + 0*z = **2x**
  - **y'** = 0*x + 3*y + 0*z = **3y**
  - **z'** = 0*x + 0*y + 5*z = **5z**

19

Using matrices: scaling

- A transformation changes a vector into another vector
- We can represent this change using a **matrix**
- Example, scale (x,y,z) by (2,3,5):

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 3 & 0 \\ 0 & 0 & 5 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Multiply elements row by column:
  - **x'** = 2*x + 0*y + 0*z = **2x**
  - **y'** = 0*x + 3*y + 0*z = **3y**
  - **z'** = 0*x + 0*y + 5*z = **5z**

20

MANCHESTER
1824

Using matrices: rotation (about Z)

- Example, rotate (x,y,z) about Z axis by θ:

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

- Multiply elements row by column:
  - **x′** = cosθ*x - sinθ*y + 0*z = **xcosθ - ysinθ**
  - **y′** = sinθ*x + cosθ*y + 0*z = **xsinθ + ycosθ**
  - **z′** = 0*x + 0*y + 1*z = **z**

MANCHESTER
1824

Using matrices: translation

- Example, translate (x,y,z) by (tx,ty,tz) :

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} ? & ? & ? \\ ? & ? & ? \\ ? & ? & ? \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

- Q: What should we put in the matrix?
- A: We can’t do it !!
- What’s the solution then?

MANCHESTER 1824

Using matrices: translation

- To incorporate translation, we have to add an **extra row and column** to the matrix, and an **extra term** to our coordinates:

$$\begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & tx \\ 0 & 1 & 0 & ty \\ 0 & 0 & 1 & tz \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This may seem like an arbitrary "fix", but it goes deeper.

And later we will see a use for the new bottom row of the matrix, for doing projection from 3D to 2D.

- Multiply elements row by column:
  - **x'** = 1*x + 0*y + 0*z + tx*1= **x + tx**
  - **y'** = 0*x + 1*y + 0*z + ty*1= **y + ty**
  - **z'** = 0*x + 0*y + 1*z + tz*1= **z + tz**
  - **w'** = 0*x + 0*y + 0*z + 1*1= **1**

23

MANCHESTER 1824

Homogeneous coordinates

- In order to use a consistent matrix representation for all kinds of linear transformations, we've had to add an extra coordinate, **w**, to our usual 3D coordinate (x,y,z).
- This (x,y,z,w) form is called "homogeneous coordinates"
- But where is this **w**? Is it in a 4^{th} spatial dimension?
- **Yes it is!**
- Unfortunately the mathematical details are beyond the scope of this course
- Usually, w=1, and we just ignore it
- When it is not, we need to "normalise"… see later, when we cover perspective

August Möbius, 1790-1868, inventor of homogeneous coordinates

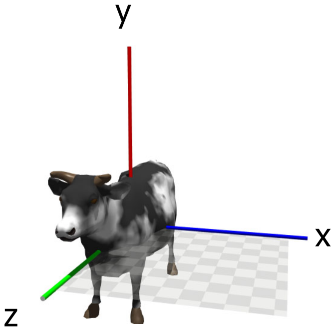
24

MANCHESTER 1824

Scale matrix

$$P' = T \cdot P$$

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



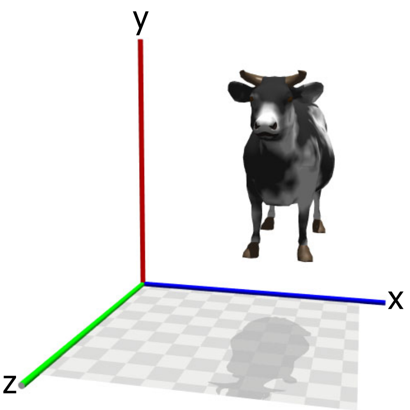
25

MANCHESTER 1824

Translation matrix

$$P' = T \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



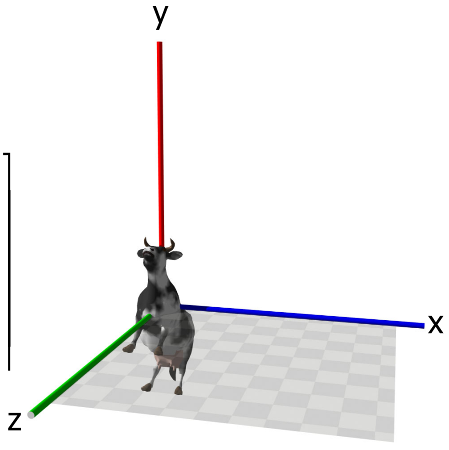
26

MANCHESTER 1824

Rotation matrix (around X axis)

$$P' = T \cdot P$$

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

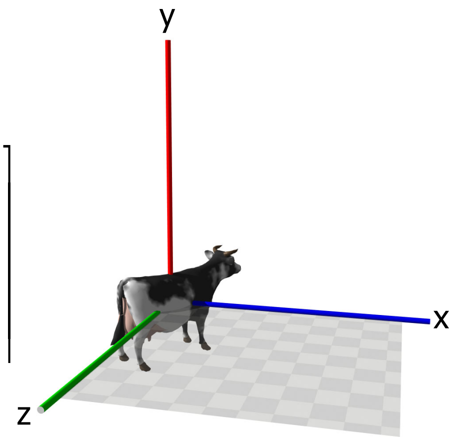


MANCHESTER 1824

Rotation matrix (around Y axis)

$$P' = T \cdot P$$

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

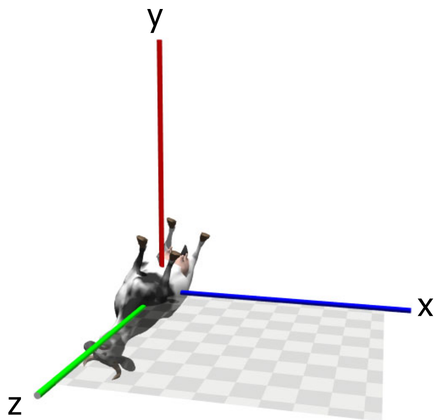


MANCHESTER 1824

Rotation matrix (around Z axis)

$$P' = T \cdot P$$

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



MANCHESTER 1824

Matrix transformations recap

- The transformation **T_1** changes point **P** to **P′**

$$P' = T_1 \cdot P$$

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ m & n & o & p \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Where the 16 values **a … p** in the matrix determine what kind of transformation it is

MANCHESTER
1824

Composing transformations

- What if we now apply a transformation $\mathbf{T_2}$ to $\mathbf{P'}$?

$$P'' = T_2 \cdot P'$$
$$P'' = T_2 \cdot T_1 \cdot P$$

- We can apply this double transformation to **P** in one go, if we multiply the matrices $\mathbf{T_1}$ and $\mathbf{T_2}$ together to obtain the **composite** transformation $\mathbf{T_C}$

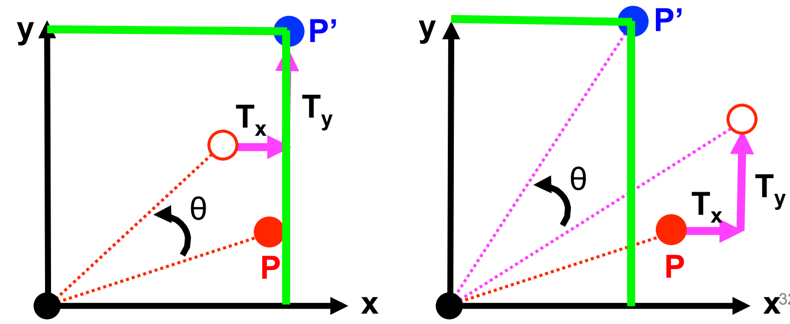
$$T_C = T_2 \cdot T_1$$
$$P'' = T_C \cdot P$$

MANCHESTER
1824

Non-commutativity, or "order matters"

- Matrix multiplications (and therefore transformations) are in general non-commutative
- Given two matrices $\mathbf{M_1}$ and $\mathbf{M_2}$, $\mathbf{M_1} * \mathbf{M_2} \neq \mathbf{M_2} * \mathbf{M_1}$

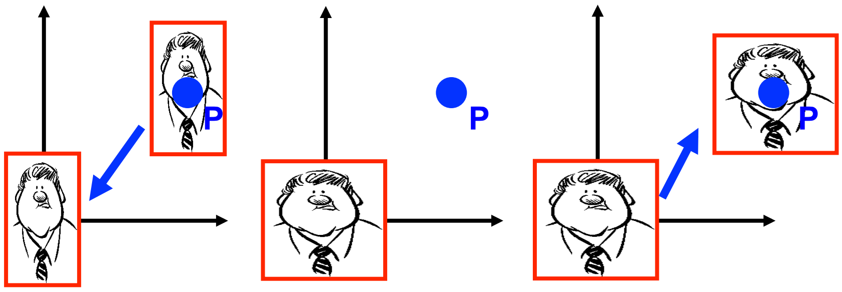
In the left example, **P** is first rotated by θ, and then shifted by (Tx,Ty). In the right example, **P** is first shifted by (Tx,Ty) and then rotated by θ. The results are different. Performing transformations in the correct order is crucial.



MANCHESTER 1824

Composite transformations, 2D example

- How to scale an object about an arbitrary point **P**, in 2D?
- We split the operation into three simpler transformations:



M_1: Translate to origin

M_2: Scale about the origin

M_3: Translate back

33

MANCHESTER 1824

Composite transformations, 2D example

- We want to scale an object by (sx, sy) about an arbitrary 2D point **P** (px, py).
- We split this into simpler steps:
  - Step 1: Construct the translation matrix **M_1** which shifts the object to the origin, by (–px, –py)
  - Step 2: Construct the matrix **M_2** which scales the object by (sx, sy) with respect to the origin
  - Step 3: Construct the translation matrix **M_3** which shifts the object back by (px, py)
- The composite transformation is $M_3 \cdot M_2 \cdot M_1$
- Note the ORDER: **M_1** first, then **M_2** then **M_3**
- The **key** to this process is in Step 3, where matrix **M_3** **undoes** the effect of matrix **M_1**

34

MANCHESTER 1824

Undoing a transformation

- Matrix **A**: shift by (Tx, Ty, Tz): $\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- Matrix **B**: shift by (-Tx, -Ty, -Tz): $\begin{bmatrix} 1 & 0 & 0 & -T_x \\ 0 & 1 & 0 & -T_y \\ 0 & 0 & 1 & -T_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$
- Matrix product of **A** and **B**: $\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$ identity matrix, **I**
- **A** and **B** are called **inverses**
  - So multiplying a point **P** by **A**, then **B**, has no effect on **P**

35

MANCHESTER 1824

Matrix inverses

- Two matrices **A** and **B** are said to be inverses of each other if: **A** x **B** = **I**, where **I** is the identity matrix
- For a matrix **M**, we write its inverse as $\mathbf{M}^{-1}$
- So, **M** x $\mathbf{M}^{-1}$ = **I**
- In other words, if a matrix **M** does some transformation on a point **P**, $\mathbf{M}^{-1}$ undoes it, restoring **P**
- Given **M**, there are algorithms for computing $\mathbf{M}^{-1}$
- **BUT**, not all matrices actually have an inverse!
  - Example: how can you undo a transformation that makes all y-coordinates 0? The original information has been destroyed…

36

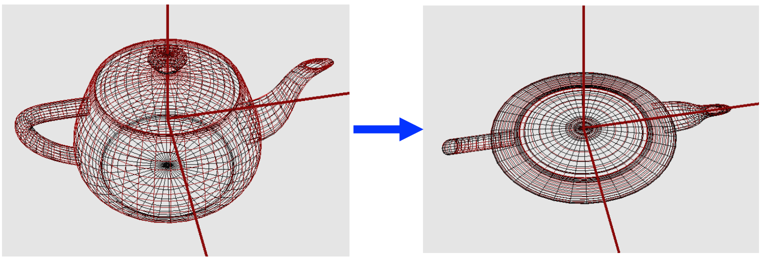
MANCHESTER 1824

Non-invertible transformations

- Example: this scale transformation will set all y-coordinates to 0:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

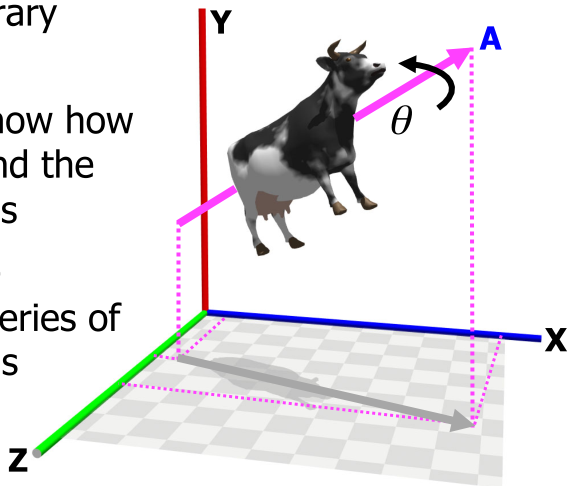
This matrix has no inverse – it is "singular". In general, a transformation is singular if it throws away information



MANCHESTER 1824

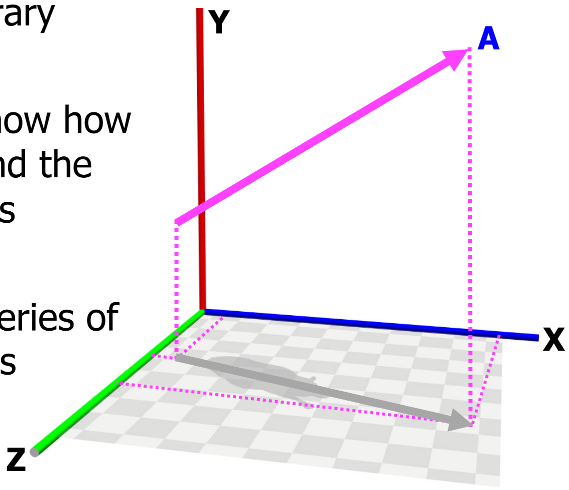
Composite transformations, 3D example

- We want to rotate by θ about an arbitrary 3D vector **A**
- But we only know how to rotate around the X, Y and Z axes
- We reduce the problem to a series of transformations



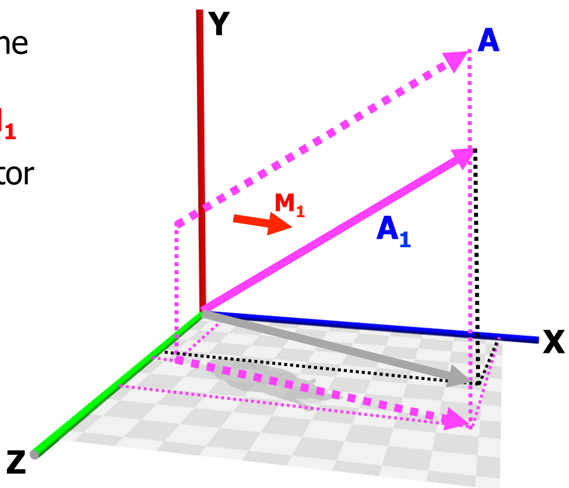
Composite transformations, 3D example

- We want to rotate by θ about an arbitrary 3D vector **A**
- But we only know how to rotate around the X, Y and Z axes
- We reduce the problem to a series of transformations
- (let's remove the cow)



Composite transformations, 3D example

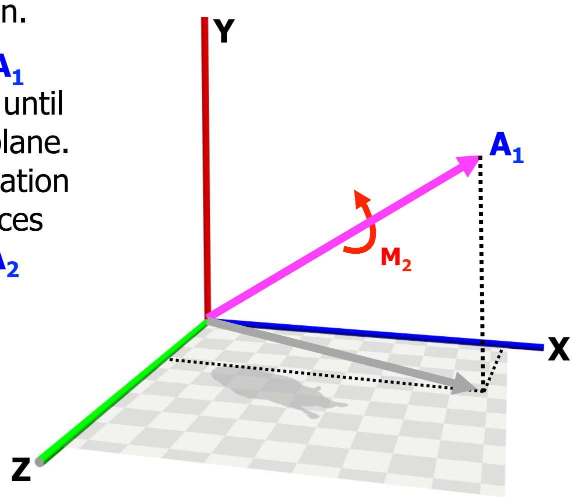
- **Step 1:** We shift vector **A** until it passes through the origin. This is transformation M_1
- Call this new vector A_1



MANCHESTER 1824

Composite transformations, 3D example

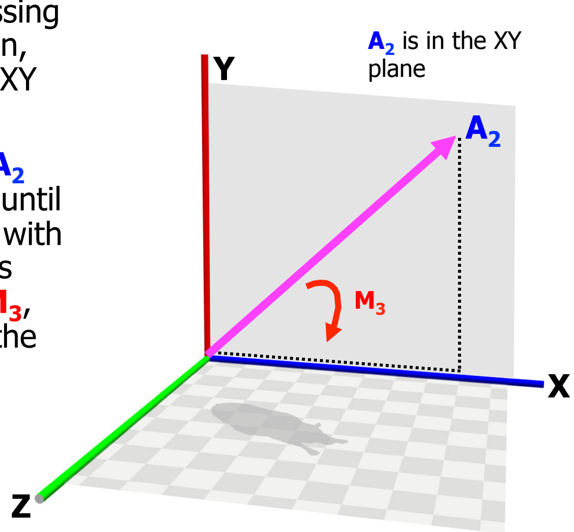
- So now A_1 is passing through the origin.
- **Step 2:** Rotate A_1 about the X-axis until it lies in the XY plane. This is transformation M_2, and it produces the new vector A_2



MANCHESTER 1824

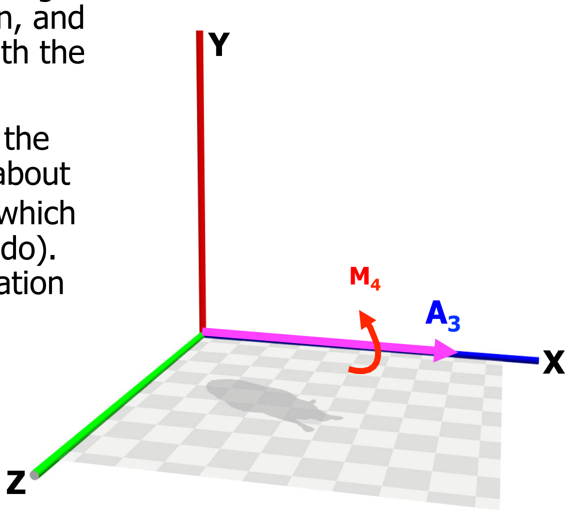
Composite transformations, 3D example

- So now A_2 is passing through the origin, and it lies in the XY plane.
- **Step 3:** Rotate A_2 about the Z axis until it it is coincident with the X-axis. This is transformation M_3, and it produces the new vector A_3



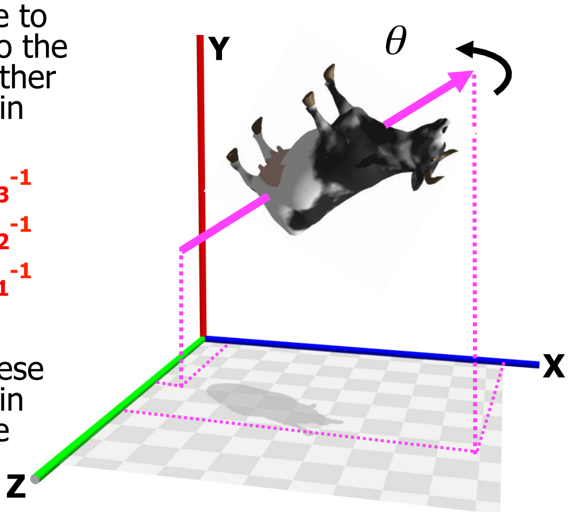
Composite transformations, 3D example

- So now $\mathbf{A_3}$ is passing through the origin, and it is coincident with the X-axis.
- **Step 4:** Perform the desired rotation about the X-axis by θ (which we know how to do). This is transformation $\mathbf{M_4}$



Composite transformations, 3D example

- So that's the rotation done. All we have to do now is to undo the effect of all the other transformations, in reverse order
- **Step 5:** apply $\mathbf{M_3}^{-1}$
- **Step 6:** apply $\mathbf{M_2}^{-1}$
- **Step 7:** apply $\mathbf{M_1}^{-1}$
- If we apply all these transformations, in order, we achieve what we want



MANCHESTER 1824

Rotation about an arbitrary 3D axis

- Summary of the steps we've gone through:
  1. Construct the matrix M_1 which translates **A** so it passes through the origin. The new vector is A_1.
  2. Construct M_2 which rotates A_1 about the X-axis (although we could use a different axis), mapping it into the XY plane. The new vector is A_2.
  3. Construct M_3, which rotates A_2 about the Z-axis, mapping it onto the X-axis. The new vector is A_3.
  4. Construct M_4, which applies the required rotation by θ about the X-axis.
  5. Construct the inverse matrices, to undo the effects of M_3, M_2 and M_1.
- The entire transformation is thus:
  - $P' = M_1^{-1} \cdot M_2^{-1} \cdot M_3^{-1} \cdot M_4 \cdot M_3 \cdot M_2 \cdot M_1 \cdot P$

MANCHESTER 1824

Transformations in OpenGL (1)

- OpenGL maintains two transformation matrices internally:
  - the **"modelview"** matrix, used for transforming the geometry you draw, and specifying the camera
  - the **"projection matrix"**, used for controlling the way the camera image is projected onto the screen (see later)
- Every 3D point you ask OpenGL to draw is automatically transformed by these two matrices before it is drawn (and you cannot prevent this happening)
  - P_{drawn} = **ProjectionMatrix** x **ModelviewMatrix** x $P_{specifed}$
- For full details, see Chapter 5 of the OpenGL manual.

MANCHESTER 1824

Transformations in OpenGL (2)

- OpenGL provides functions for easily dealing with transformations. Here are some:

```
glTranslatef(tx, ty, tz)
glScalef(sx, sy, sz)
glRotatef(theta, rx, ry, rz)
```

OpenGL

- When we call one of these functions, OpenGL creates a corresponding temporary matrix **TMP**, and then multiples the **modelview** matrix by **TMP**, and then throws away **TMP**

MANCHESTER 1824

Transformations in OpenGL (3)

- An example. We want to **first** rotate and **then** shift the teapot.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity(); // M= identity matrix (I)
glTranslatef(tx, ty, tz);
// OpenGL computes temp translation matrix T,
// then sets M= M x T, so now M is T
glRotatef(theta, 0.0, 1.0, 0.0);
// OpenGL computes temp rotation matrix R,
// then sets M= M x R, so M is now T x R
glutWireTeapot(1.0);
```

- Notice the **order** we call the functions in… it's the **reverse** of how we would write it down logically.

Transformations in OpenGL (4)

- What if we want a series of steps, as we saw earlier?
- Sometimes there are OpenGL functions which come to our rescue. For example, **`glRotatef()`** will conveniently compute a matrix for rotation of angle θ about the vector (x, y, z) which passes through the origin.

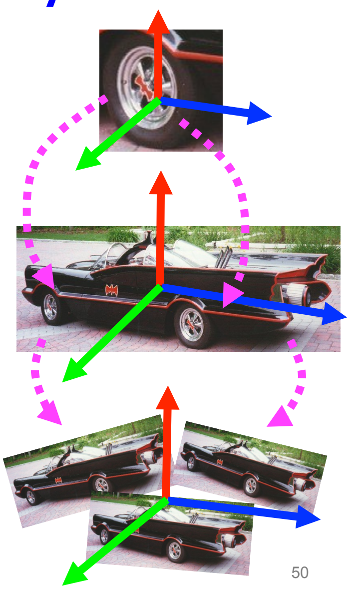
OpenGL

```
glRotatef(GLfloat theta,
          GLfloat x, GLfloat y,
          GLfloat z);
```

- Therefore, if we call this transformation **R**, we can express our previous rotation-about-arbitrary-vector example as $\mathbf{M_1^{-1} \cdot R \cdot M_1}$
- There are OpenGL functions for loading your own matrix from the modelview or projection matrices, and for multiplying them together. But in practice, it's not necessary to use these much.

Model and world coordinate systems

- Often an object is defined in a local **modelling coordinate system**. E.g. modelling a car wheel with an origin at the wheel centre.
- **Modelling transformations** are used to **instance** multiple copies of an object in the scene, e.g. translate and rotate the wheels onto a car body
- The entire car may then have further transformations applied, like translation to simulate its movement.
- A global **world coordinate system** is used to specify the position of objects in the entire scene.



MANCHESTER
1824

Reference: Useful vector geometry

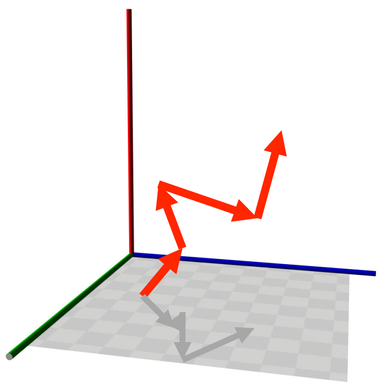
- Vectors provide a very convenient way of thinking about many of the manipulations we might want to perform on an object in 3D space
- In fact, it's the only sensible way to work (and essential for rendering, as we shall see later)
- Understanding a small amount of vector maths goes a long way in 3D graphics…
  - Addition and Subtraction
  - Scalar multiplication
  - Vector normalization
  - Dot product
  - Cross product

MANCHESTER
1824

Vector addition

- To add two vectors of the same order, add the components…

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} + \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 + x_2 \\ y_1 + y_2 \\ z_1 + z_2 \\ 1 \end{bmatrix}$$



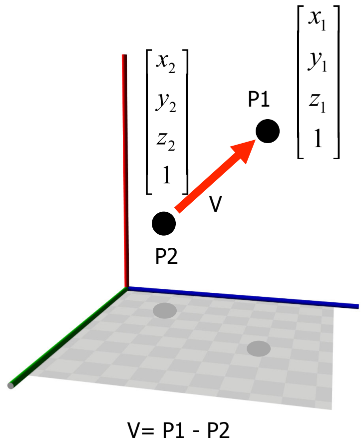
- Why is this useful…?
- … moves a point through space in a known direction

MANCHESTER 1824

Vector subtraction

- To subtract two vectors of the same order, subtract the components…

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} - \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} x_1 - x_2 \\ y_1 - y_2 \\ z_1 - z_2 \\ 1 \end{bmatrix}$$



- Why is this useful…?
- … represents 'a line' between two points

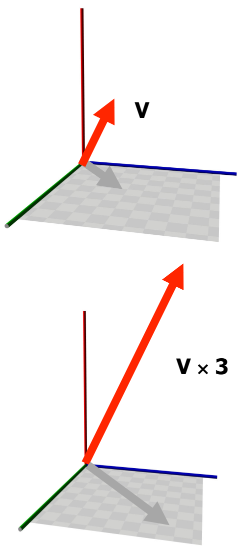
53

MANCHESTER 1824

Multiplication by a scalar

- Multiply the individual components by a scalar ***C***

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \times C = \begin{bmatrix} x_1 \times C \\ y_1 \times C \\ z_1 \times C \\ 1 \end{bmatrix}$$



- Why is this useful…?
- … moves a point along a vector by a given amount

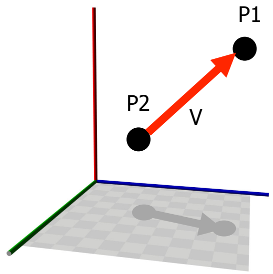
54

MANCHESTER 1824

Vector magnitude

- Gives the 'length' or size of a vector

$$V = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \qquad |V| = \sqrt{x^2 + y^2 + z^2}$$



- If we have 'a line' joining two points, the magnitude of the vector between them represents their distance in 3D space

V= P1 - P2

Distance from P1 to P2 is |V|

55

MANCHESTER 1824

Vector normalization

- **Normalization** is the process of taking an arbitrary (but non-zero) vector **V**, and converting it into a vector $\hat{\mathbf{V}}$ (V-hat) **of length 1,** which points in the same direction
- Calculate the length L of **V**, and divide its *x*, *y* and *z* components by this value
- Essential operation in rendering

$$\mathbf{V} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

$$L = \sqrt{x^2 + y^2 + z^2}$$

$$\hat{\mathbf{V}} = \begin{bmatrix} x/L \\ y/L \\ z/L \\ 1 \end{bmatrix}$$

56

Vector multiplication

- There are two ways of multiplying vectors
- One results in a **scalar value**, and is called the **dot product** (aka "inner product")
- The other results in a **vector**, and is called the **cross product** (aka "outer product")

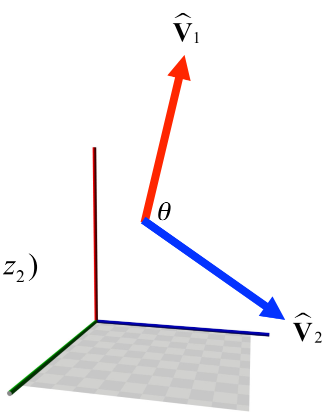
- Both are essential operations in 3D graphics

57

The Dot Product

- is the scalar product of the individual components

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \bullet \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = (x_1 \times x_2) + (y_1 \times y_2) + (z_1 \times z_2)$$



$$\cos\theta = \hat{\mathbf{V}}_1 \bullet \hat{\mathbf{V}}_2$$

- For normalized vectors, their dot product is the cosine of the angle between them
- Essential for rendering

58

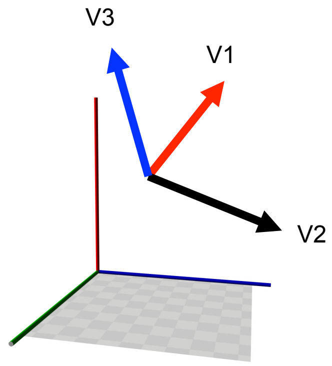
MANCHESTER 1824

The Cross Product

- is a vector, defined as follows:

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \\ 1 \end{bmatrix} \times \begin{bmatrix} x_2 \\ y_2 \\ z_2 \\ 1 \end{bmatrix} = \begin{bmatrix} y_1 \times z_2 - z_1 \times y_2 \\ x_1 \times z_2 - z_1 \times x_2 \\ x_1 \times y_2 - y_1 \times x_2 \\ 1 \end{bmatrix}$$

- For two vectors, their cross product is a third vector **perpendicular** to them both (forming a right handed system)



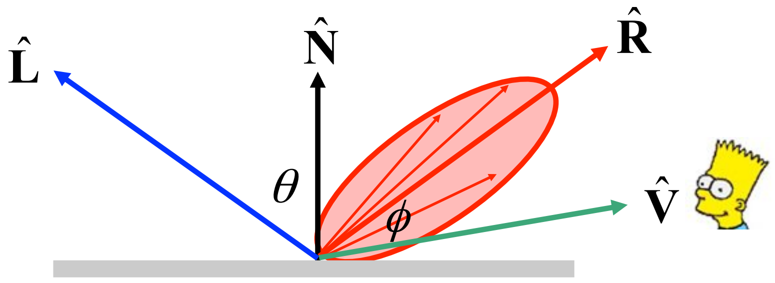
V3 = V1 x V2

Note: the version of this slide in the paper handout is wrong (apologies). This is correct.

MANCHESTER 1824

Vector geometry is essential

- All these properties of vectors are essential in 3D graphics:
  - for defining and manipulating geometry
  - for specifying and evaluating rendering



- There are many vector manipulation libraries available that hide the underlying maths and make vector manipulation easy