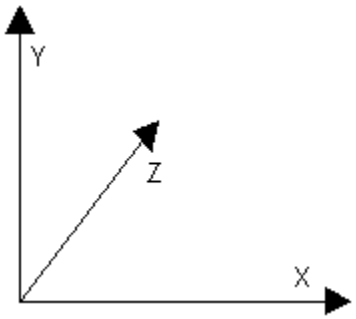
Introduction to Computer Graphics
(C S 6 0 2)
Lecture 17
3D Transformations I

17.1 Definition of a 3D Point

A point is similar to its 2D counterpart; we simply add an extra component, Z, for the 3rd axis:



Points are now represented with 3 numbers: <x, y, z>. This particular method of representing 3D space is the "left-handed" coordinate system. In the left-handed system the x axis increases going to the right, the y axis increases going up, and the z axis increases going into the page/screen. The right-handed system is the same but with the z-axis pointing in the opposite direction.

17.2 Distance between Two 3D Points

The distance between two points <Ax,Ay,Az> and <Bx,By,Bz> can be found by again using the Pythagoras theorem:

```
dx = Ax-Bx
dy = Ay-By
dz = Az-Bz
distance = sqrt(dx*dx + dy*dy + dz*dz)
```

17.3 Definition of a 3D Vector

Like it's 2D counterpart, a vector can be thought of in two ways: either a point at <x,y,z> or a line going from the origin <0,0,0> to the point <x,y,z>.

3D Vector addition and subtraction is virtually identical to the 2D case. You can add a 3D vector <vx,vy,vz> to a 3D point <x,y,z> to get the new point <x',y',z'> like so:

```
x' = x + vx
y' = y + vy
z' = z + vz
```

Vectors themselves can be added by adding each of their components, or they can be multiplied (scaled) by multiplying each component by some constant k (where k <> 0). Scaling a vector by 2 (say) will still cause the vector to point in the same direction, but it will now be twice as long. Of course you can also divide the vector by k (where k <> 0) to get a similar result.

To calculate the length of a vector we simply calculate the distance between the origin and the point at <x, y, z>:

```
Length = | <x,y,z> - <0,0,0> |
       = sqrt( (x-0)*(x-0) + (y-0)*(y-0) + (z-0)*(z-0) )
       = sqrt(x*x + y*y + z*z)
```

17.4 Unit Vector

Often in 3D computer graphics you need to convert a vector to a unit vector, ie a vector that points in the same direction but has a length of 1.

This is done by simply dividing each component by the length:

$$\text{Let } \langle x,y,z \rangle \text{ be our vector, length} = \sqrt{x*x + y*y + z*z}$$

$$\text{Unit vector} = \frac{\langle x,y,z \rangle}{\text{length}} = \left| \frac{x}{\text{length}}, \frac{y}{\text{length}}, \frac{z}{\text{length}} \right|$$

(Where length = |<x,y,z>|)

Note that if the vector is already a unit vector then the length will be 1, and the new values will be the same as the old.

17.5 Definition of a Line

As in 2D, we can represent a line by it's endpoints (P1 and P2) or by the parametric equation:

```
P = P1 + k * (P2-P1)
```

Where k is some scalar value between 0 and 1

17.6 Transformations:

A static set of 3D points or other geometric shapes on screen is not very interesting. You could just use a paint program to produce one of these. To make

your program interesting, you will want a dynamic landscape on the screen. You want the points to move in the world coordinate system, and you even want the point-of-view (POV) to move. In short, you want to model the real world. *The process of moving points in space is called transformation*, and can be divided into *translation, rotation and other kind of transformations*.

17.7 Translation

Translation is used to move a point, or a set of points, linearly in space, for example, you may want to move a point "3 meters east, -2 meters up, and 4 meters north." Looking at this textual description, you might think that this looks very much like a Point3D, and you would be close. But the above does not require one critical piece of information: it does not reference the origin. The above only encapsulates direction and distance, not an absolute point in space. This called a vector and can be represented in a structure identical to Point3D:

```
struct Vector3D
        float x;        distance along x axes
        float y;        distance along y axes
        float z;        distance along z axes
end struct
```

17.8 Vector Addition

You translate a point by adding a vector to it; you add points and vectors by adding the components piecewise:

```
Point3D point = {0, 0, 0}
Vector3D vector = {10, -3, 2.5 }
```

Adding vector to point

```
point.x = point.x + vector.x;
point.y = point.y + vector.y;
point.z = point.z + vector.z;
```

Point will be now at the absolute point < 10,-3 2.5>. you could move it again:

```
point.x = point.x + vector.x;
point.y = point.y + vector.y;
point.z = point.z + vector.z;
```

And point would now be at the absolute point <20, -6, 5>.

In pure mathematical sense, you cannot add two points together – such an operation makes no sense (what is Lahore plus Karachi?). However, you can subtract a point from another in order to uncover the vector that would have to be added to the first to translate it into the second:

```
Point3D p1,p2
Vector3D v;

Set p1 and p2 to the desired points
v.x = p2.x – p1.x
v.y = p2.y – p1.y
v.z = p2.z – p1.z
```

Now you can add v to p1, you would translate it into the point p2.

The following lists the operations you can do between points and vectors:

```
point – point  => vector
point + point = point - ( - point) => vector
vector – vector => vector
vector + vector => vector
point – vector = point + (-vector) => point
point + vector => point
```

17.9 Multiplying: Scalar Multiplication

Multiplying a vector by a scalar (a number with no units), and could be coded with:

```
Vector.x =  Vector.x  * scalarValue
Vector.y =  Vector.y  * scalarValue
Vector.z =  Vector.z  * scalarValue
```

If you had a vector with a length of 4 and multiplied it by 2.5, you would end up with a vector of length 10 that points in the same direction the original vector pointed. If you multiplied by -2.5 instead, you would still end up with a vector of length 10; but now it would be pointing in the opposite direction of the original vector.

17.10 Multiplying: Vector Multiplication

You can multiply with vectors two other ways; both involve multiplying a vector by a vector.

17.11 Dot Product

The dot product of two vectors is defined by the formula:
Vector A, B

A * B = A.x * B.x + A.y * B.y + A.z * B.z

The result of a dot product is a number and has units of A's units times B's units. Thus, if you calculate the dot product for two vectors that both use feet for units, your answer will be in square feet. However, in 3D graphics we usually ignore the units and just treat it like a scalar.
Consider the following definition of the dot product that is used by physicists (instead of mathematicians):

A * B = |A| * |B| * cos(theta)

Where theta is the angle between the two vectors

Remember that |v| represents the length of vector V and is a non-negative number; we can replace the vector lengths above and end up with:

K = |A| * |B| (therefore k > = 0)

A * B = K * cos (theta)

Therefore:

A * B => cos(theta)

Where "=>" means "directly correlates to." Now, if you remember, the cos(theta) function has the following properties:

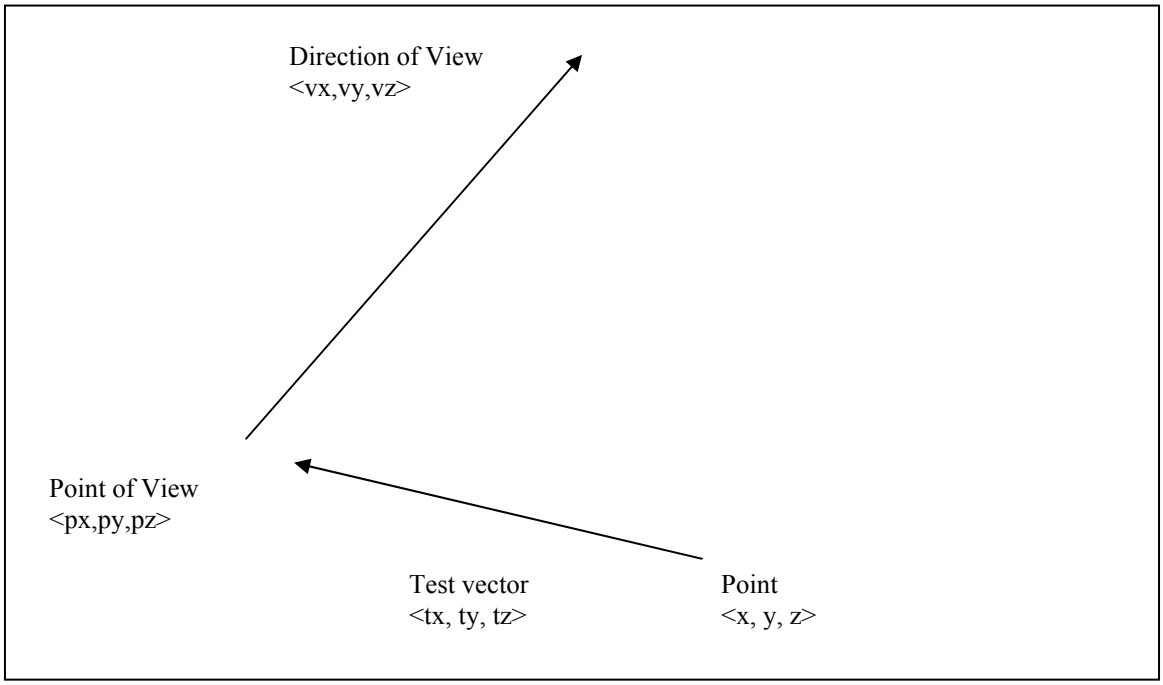
cos(theta) > 0 iff theta is less than 90 degrees or greater than 270 degrees
cos(theta) < iff theta is greater than 90 degrees and less than 270 degrees
cos(theta) = 0 iff theta is 90 degrees or 270 degrees

We can extend this to the dot product of two vectors, since it directly correlates to the angle between the two vectors:

A*B > 0 iff the angle between them is less than 90 or greater than 270 degrees
A*B < 0 iff the angle between them is greater than 90 and less than 270 degrees
A*B = 0 iff the angle between them is 90 or 270 degrees (they are orthogonal).

17.12 Use of Dot Product

Assume you have a point of view at < px,py,pz>. It is looking along the vector <vx,vy,vz>, and you have a point in space <x,y,z> you want to know if the point–of-view can possible see the point, of if the point is "behind " the POV, as shown in figure.




```
Point3D pov;
Vector3D povDir;
Point3D test;
Vector3D vTest
float dotProduct;
vTest.x = pov.x – test.x;
vTest.y = pov.y – test.y;
vTest.z = pov.z – test.z;

dotProduct == vTest.x*povDir.x  + vTest.y*povDir.y + vTest.z * povDir.z;

if(dotProduct > 0)
        point is "in front of " POV
else if (dotProduct < 0)
```

point is “behind” POV
else
point is orthogonal to the POV direction

17.13 Cross Product

Another kind of multiplication that you can do with vectors is called the cross product this is defined as:
Vector A, B

A X B = < A.y * B.z – A.z * B.y, A.z * B.x – A.x * B.z, A.x * B.y – A.y * B.x >

For physicists:

|A x B| = |A| * |B| sin(theta)

Where theta is the angle between the two vectors.

The above formula for A x B came from the determinate of order 3 of the matrix:

| X Y Z |
|A.x A.y A.z|
|B.x B.y B.z|

17.14 Transformations

The process of moving points in space is called transformation.

17.15 Types of Transformation

There are various types of transformations as we have seen in case of 2D transformations. These include:

a) Translation
b) Rotation
c) Scaling
d) Reflection
e) Shearing

a) Translation

Translation is used to move a point, or a set of points, linearly in space. Since now we are talking about 3D, therefore each point has 3 coordinates i.e. x, y and z. similarly, the translation distances can also be specified in any of the 3 dimensions. These Translation Distances are given by tx, ty and tz.
For any point P(x,y,z) after translation we have P′(x′,y′,z′) where

x′ = x + tx ,
y′ = y + ty ,
z′ = z + tz
and (tx, ty , tz) is Translation vector

Now this can be expressed as a single matrix equation:

$$P' = P + T$$

Where:

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \qquad P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} \qquad T = \begin{bmatrix} t_x \\ t_y \\ t_z \end{bmatrix}$$

3D Translation Example

We may want to move a point “3 meters east, -2 meters up, and 4 meters north.” What would be done in such event?

Steps for Translation

Given a point in 3D and a translation vector, it can be translated as follows:

Point3D point = (0, 0, 0)
Vector3D vector = (10, -3, 2.5)

Adding vector to point

point.x = point.x + vector.x;
point.y = point.y + vector.y;
point.z = point.z + vector.z;

And finally we have translated point.

Homogeneous Coordinates

Analogous to their 2D Counterpart, the homogeneous coordinates for 3D translation can be expressed as :

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

Abbreviated as:

$$P' = T(tx, ty, tz).\ P$$

On solving the RHS of the matrix equation, we get:

$$\begin{bmatrix} x' \\ y' \\ z' \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

Which shows that each of the 3 coordinates gets translated by the corresponding translation distance.