Basic elements of 3-D coordinate systems and linear algebra

Coordinate systems are used to assign numeric values to locations with respect to a particular frame of reference commonly referred to as the *origin*. The number of dimensions in the coordinate system is equal to the number of perpendicular (orthogonal) axes and is also the number of values needed to specify a location with respect to the origin.

**One dimension: the line**

![One dimension diagram]

**Two dimensions: the plane**

![Two dimensions diagram]
**Three dimensions: the universe as we perceive it**

(A right handed coordinate system is shown. In a left handed system the direction of the positive z axis is reversed. )

![Diagram of 3D space showing the coordinate system and a point labeled (4.5, 2, -3).](image)

**Points in 3-D space**

The location of a *point P* in 3-D Euclidean space is given by a triple \((p_x, p_y, p_z)\)

The \(x\), \(y\), and \(z\) coordinates specify the distance you must travel in directions parallel to the \(x\), \(y\), and \(z\) axes starting from the origin \((0, 0, 0)\) to arrive at the point \((p_x, p_y, p_z)\)
Vectors: Some physical properties, such as temperature or area, are given completely by their magnitude, and so only need a number (called a scalar) to represent them. But there are other physical quantities, such as force, velocity, or acceleration, for which we must know direction as well as size, or magnitude, in order to work with them. It is often very helpful to represent such quantities by directed lines called vectors. Because vectors carry the physical information of both magnitude and direction, using them gives us a very neat way of handling these quantities.

- Two vectors are equal if and only if they are equal in both magnitude and direction. So \( \mathbf{a} \) is not equal to \( \mathbf{b} \) although they are the same length, because they are in different directions. However, the two vectors marked \( \mathbf{c} \) are equal.
- If \( \mathbf{c} \) is a vector, then \( - \mathbf{c} \) is defined as having the same magnitude but the reverse direction to \( \mathbf{c} \). Subtracting \( \mathbf{c} \) is the same as adding \( - \mathbf{c} \).
- Multiplying a vector by a number or scalar just has the effect of changing its scale. So, for example, \( 2\mathbf{a} \) would be twice as long as \( \mathbf{a} \) but in the same direction as \( \mathbf{a} \).

- The two vectors labelled \( \mathbf{c} \) are defined to be equal although we need a shift to move them exactly on top of each other. Vectors that can be shifted about are called free vectors.

- There is one important situation in which we can't shift the vectors around: displacement from a given fixed point such as the origin is a vector quantity, but the vector is tied to the fixed point; the vector \( \mathbf{r} \) in the drawing above is called the position vector of the point \( \mathbf{P} \) from the fixed origin \( \mathbf{O} \), and describes the displacement of \( \mathbf{P} \) from \( \mathbf{O} \). It must run from \( \mathbf{O} \) to \( \mathbf{P} \) and can't be shifted around.

**Vectors in 3-D space**

A vector in 3-D space is sometimes called a directed distance because it represents both

- a direction and
- a magnitude or distance

In this context, the triple \( (p_x, p_y, p_z) \) can also be considered to represent

- the direction from the origin \((0, 0, 0)\) to \((p_x, p_y, p_z)\) and
- its length \( \sqrt{p_x^2 + p_y^2 + p_z^2} \) is the Euclidean (straight line) distance from the origin to \((p_x, p_y, p_z)\).
Points and vectors

Two points in 3-D space implicitly determine a vector pointing from one to the other. Given two points P and Q in 3-D Euclidean space, the vector

\[ R = P - Q = (p_x - q_x, p_y - q_y, p_z - q_z) \]

represents the direction from Q to P. Note that the direction is a signed quantity. The direction from P to Q is the negative of the direction from Q to P.

Its length, as defined above, is the distance between P and Q. The distance from P to Q is always the same as the distance from Q to P.

Example: Let Q = (8, 6, 5) and P = (3, 2, 0).

Then the vector direction from Q to P is: \((3 - 8, 2 - 6, 0 - 5) = (-5, -4, -5)\)

The vector direction from P to Q is \((5, 4, 5)\)

The distance between Q and P is: \(\sqrt{25 + 16 + 25} = \sqrt{66} = 8.12\).

The following link has more explanations and a few videos demonstrating vector addition and subtraction in 3D programming:

The geometric interpretation of vector arithmetic

Here we work with 2 dimensional vectors to simplify the visual interpretation, but in 3-d the principles are the same.

Since we can think of vectors as displacements or journeys, to add two vectors, we just need to find the single displacement which gives the same result as doing the two displacements separately.

(For example, if a boat sets a course to move with velocity $\mathbf{P}$ in water flowing with velocity $\mathbf{Q}$, then it will actually have a Resultant velocity of $\mathbf{P} + \mathbf{Q}$.)

$\mathbf{P} = (5, 1) \Rightarrow +5 \text{ in the } x \text{ direction and then } +1 \text{ in the } y \text{ direction}$

$\mathbf{Q} = (2, 4) \Rightarrow +2 \text{ in the } x \text{ direction and } +4 \text{ in the } y \text{ direction.}$

$\mathbf{R} = \mathbf{P} + \mathbf{Q} = (7, 5)$

$\mathbf{P} = \mathbf{R} + (−\mathbf{Q}) = \mathbf{R} - \mathbf{Q}$
Useful operations on vectors:

We define the **sum** of two vectors $P$ and $Q$ as the component-wise sums:
\[
R = P + Q = (p_x + q_x, \ p_y + q_y, \ p_z + q_z)
\]
\[
(3, \ 4, \ 5) + (1, \ 2, \ 6) = (4, \ 6, \ 11)
\]

The **difference** of two vectors is computed as the component-wise differences:
\[
R = P - Q = (p_x - q_x, \ p_y - q_y, \ p_z - q_z)
\]
\[
(3, \ 4, \ 5) - (1, \ 2, \ 6) = (2, \ 2, \ -1)
\]

We also define multiplication (or **scaling**) of a vector by a scalar number $a$

- stretches out the vector by a factor of $a$; makes it that many times longer
- if $a$ is negative, the vector changes direction
\[
S = aP = (ap_x, \ ap_y, \ ap_z)
\]
\[
3 \times (1, \ 2, \ 3) = (3, \ 6, \ 9)
\]

The **length** (magnitude) of a vector $P$ is a scalar whose value is denoted:
\[
|| P || = \sqrt{p_x^2 + p_y^2 + p_z^2}
\]
\[
||(3, \ 4, \ 5)|| = \sqrt{9 + 16 + 25} = \sqrt{50}
\]

A **unit vector** is a vector whose length is 1. Therefore an arbitrary vector $P$ may be converted to a unit vector by scaling it by $1 / (\text{its own length})$. Here $U$ is a unit vector in the same direction as $P$.
\[
U = \left( \frac{1}{|| P ||} \right) P
\]

The **inner product** or **dot product** of two vectors $P$ and $Q$ is a scalar number. It is computed by taking the sum of the component-wise products of the two vectors.
\[
x = P \cdot Q = (p_x q_x + p_y q_y + p_z q_z)
\]
\[
(2, \ 3, \ 4) \cdot (3, \ 2, \ 1) = 6 + 6 + 4 = 16
\]

Thus $|| P || = \sqrt{P \cdot P}$

If $U$ and $V$ are unit vectors and $q$ is the angle between them then:
\[
\cos (q) = U \cdot V = V \cdot U
\]
Representing vectors in C

There are at least two easy ways to represent a vector:

Array based representation:

We can use the \texttt{typedef} facility to create a user defined type called \texttt{vec\_t}. An instance of \texttt{vec\_t} is a three element double precision array:

\begin{verbatim}
typedef double vec_t[3];
\end{verbatim}

An instance of \texttt{vec\_t} is three element double precision array and can be created as shown.

\begin{verbatim}
vec\_t vec;
\end{verbatim}

It is understood that

\begin{verbatim}
vec[0] is the x-component (coordinate)
vec[1] is the y-component
vec[2] is the z-component
\end{verbatim}

To make this association explicit we use the \texttt{#define} facility

\begin{verbatim}
#define X 0
#define Y 1
#define Z 2
\end{verbatim}

We can then create an instance of the vector (11, 2, -4) as shown:

\begin{verbatim}
vec\_t vec;
vec[X] = 11;
vec[Y] = 2;
vec[Z] = -4;
\end{verbatim}
Structure based representation

We can also define a structured type in which the elements are explicitly named

```c
typedef struct vec_type
{
    double x;
    double y;
    double z;
} vec_t;
```

```c
vec_t vec;
```

In this representation, it is explicit that

- `vec.x` is the x-component
- `vec.y` is the y-component
- `vec.z` is the z-component

Religious wars have been fought over which is “correct”. We will refuse to engage in the war, but we will use the structure based approach in this course.

Because elements of both the structure and the array are guaranteed to be packed into adjacent memory elements, it is possible to cheat and use either array or structure notation.

```c
vec_t v = {1.0, 2.0, 3.0};
double *w = (double *)&v;

printf("%lf %lf %lf\n",  v.x, v.y, v.z);
printf("%lf %lf %lf\n",  w[0], w[1], w[2]);
```

```
1.000000 2.000000 3.000000
1.000000 2.000000 3.000000
```
A library for 3-D vector operations

Since the above operations will be commonly required in the raytracer, you will build a library of functions which we will call vector.h to perform them. Here are the function prototypes that must be employed.

```c
/* Scale a 3d vector */
void vec_scale(double fact, vec_t *v1, vec_t *v2);

where double fact is the scale factor
vec_t *v1 is the input vector
and vec_t *v2 is the output vector

*****************************************************************
/* Return length of a 3d vector */
double vec_len(vec_t *v1); /* Vector whose length is desired */

*****************************************************************
/* Compute the difference of two vectors */
/* v3 = v2 - v1 */
void vec_diff(vec_t *v1, vec_t *v2, vec_t *v3);

where vec_t *v1 is the subtrahend
vec_t *v2 is the minuend
and vec_t *v3 is the result

*****************************************************************
/* Compute the sum of two vectors */
/* v3 = v2 + v1 */
void vec_sum(vec_t *v1, vec_t *v2, vec_t *v3);

where vec_t *v1 is the addend
vec_t *v2 is the addend
and vec_t *v3 is the result
```
/* Return the inner product of two input vectors */
double vec_dot(vec_t *v1, vec_t *v2);

where vec_t *v1 is the input vector 1
and vec_t *v2 is the input vector 2

*****************************************************************
/* Copy one vector to another */
void vec_copy(vec_t *v1, vec_t *v2);

where vec_t *v1 is the input vector
and vec_t *v2 is the output vector

*****************************************************************
/* Construct a unit vector in direction of input */
void vec_unit(vec_t *v1, vec_t *v2);

where vec_t *v1 is the input vector
and vec_t *v2 is the output unit vector

*****************************************************************
/* Read in values of vector from file */
void vec_read(FILE *in, vec_t *v1);

*****************************************************************
/* Print values of vector to file */
void vec_print(FILE *out, char *label, vec_t *v1);

where FILE *out is the output file
char *label is the label string
and vec_t *v1 is the vector to print
Warning regarding aliased parameters

When parameters are passed using pointers, a potentially destructive phenomenon known as aliasing may occur. Here the caller of \texttt{vec\_unit()} is requesting that a vector be converted to a unit vector in place.

\begin{verbatim}
vec_unit(v1, v1);
\end{verbatim}

Now suppose the implementation of \texttt{vec\_unit()} is as follows:

\begin{verbatim}
void vec_unit(vec_t *vin, vec_t *vout)
{
    vout->x = vin->x / vec_len(vin);
    vout->y = vin->y / vec_len(vin);
    vout->z = vin->z / vec_len(vin);
}
\end{verbatim}

This looks correct and (assuming \texttt{vec\_len()}) is working properly it will work correctly as long as the parameters \texttt{vin} and \texttt{vout} point to different vectors. However, if they point to the \textit{same vector} incorrect computation will result. If \texttt{vin} and \texttt{vout} point to the same vector, the assignment

\begin{verbatim}
    vout->x = vin->x / vec_len(vin);
\end{verbatim}

also changes \texttt{vin->x} Therefore, in the subsequent steps of the computation

\begin{verbatim}
    vout->y = vin->y / vec_len(vin);
    vout->z = vin->z / vec_len(vin);
\end{verbatim}

\texttt{vec\_len()} will generally (but not always) return a \textit{different value than in the preceding step}. For the computation to work correctly, \texttt{vec\_len()} must always return the \textit{length of the original input vector}.  


A correct version of `vec_unit()`

The function can be written correctly (and more efficiently) as.

```c
void vec_unit(vec_t *vin, vec_t *vout)
{
    double scale = 1.0 / vec_len(vin);
    vec_scale(scale, vin, vout);
}
```

ALL vector functions *must* work correctly with aliased parameters.
A sample test driver for `vector.h`

```c
#include <math.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vector.h"

vec_t v1 = {3.0, 4.0, 5.0};
vec_t v2 = {4.0, -1.0, 2.0};

int main()
{
    vec_t v3;
    double v;

    vec_print(stdout, "v1", &v1);
    vec_print(stdout, "v2", &v2);

    vec_diff(&v1, &v2, &v3);
    vec_print(stdout, "v2 - v1 = ", &v3);

    v = vec_dot(&v1, &v2);
    printf("v1 dot v2 is %8.3lf \n", v);

    v = vec_len(&v1);
    printf("Length of v1 is %8.3lf \n", v);

    vec_scale(1 / v, &v1, &v3);
    vec_print(stdout, "v1 scaled by its 1/ length:", &v3);

    vec_unit(&v1, &v1);
    vec_print(stdout, "unit vector in v1 direction:", &v1);

    return(0);
}
```

```
acad/cs102/labs10/lab1 ==> a.out
v1   3.000   4.000   5.000
v2   4.000  -1.000   2.000
v2 - v1 =    1.000  -5.000  -3.000
v1 dot v2 is   18.000
Length of v1 is    7.071
v1 scaled by its 1/ length:   0.424   0.566   0.707
unit vector in v1 direction:   0.424   0.566   0.707
```
Representing \(rgb\) data

In the raytracer, we will work with three types of \(rgb\) data:

- reflective materials
- emissive lights
- pixels

We will use \(rgb\) data for all three, but will use different models for the interaction of lights with materials than for the image itself.

As in CPSC 101, for the image data used in the .ppm file, we use an unsigned character representation where 0 means black and 255 means maximal brightness.

For representing pixel values, we use:

```c
typedef struct irgb_type
{  unsigned char r;
    unsigned char g;
    unsigned char b;
} irgb_t;
```

For representing lights, reflective materials and their interactions we use:

```c
typedef struct drgb_type
{  double r;
    double g;
    double b;
} drgb_t;
```

In this representation 0.0 means black and 1.0 means maximal brightness. It is possible to produce values > 1.0 and as in CPSC 101 these must be clamped to lie within the range [0, 255.999] before converting to \(irgb\_t\).

The \(irgb\_t\) will be used only in the final image.