More Vector Functions (pgs 1-11) and the Generalized Parser (pgs 12-20)

We will add more vector functions to our vector.c file, which will be used in the near future for some of the lighting features. For future reference, the picture below shows what specular reflection and diffuse reflection are.
The **Cross Product** of two vectors

If you have 3 points on a plane, two vectors can be drawn. The cross product of these 2 vectors *will give the normal to the plane*. So, given two linearly independent (not parallel) vectors:

\[
V = (v_x, v_y, v_z) \\
W = (w_x, w_y, w_z)
\]

The **cross product** sometimes called *outer product* is a vector which is orthogonal (perpendicular to) both of the original vectors.

\[
V \times W = (v_y w_z - v_z w_y, v_z w_x - v_x w_z, v_x w_y - v_y w_x)
\]

(1, 1, 1) \times (0, -1, 0) = (1, 0, -1)

**Notes:**
The vector (0, -1, 0) is the negative *y axis*. Therefore, any vector that is perpendicular to it must lie in the *y = 0* plane. The projection of the vector (1, 1, 1) onto the *y = 0* plane is the vector (1, 0, 1). The vector (1, 0, -1) is then perpendicular to this vector and lies in the *y* = 0 plane.

In a *right-handed* coordinate system

\[
X \times Y = Z \\
Y \times Z = X \\
Z \times X = Y
\]

Right thumb (x-axis) \(\times\) forefinger (y-axis) = middle finger (z-axis).

**The vec_cross() function**

You will add the following function to your vector.c collection.

```c
/* Compute the outer product of two input vectors */
void vec_cross(
    vec_t  *v1,         /* Left input vector */
    vec_t  *v2,         /* Right input vector */
    vec_t  *v3)         /* Output vector */
{
}
```
**Projection**

Assume that $V$ and $N$ are **unit vectors**. The projection, $Q$, of $V$ **on** $N$ is shown in red. It is a vector in the same direction as $N$ but having length $\cos(\theta)$. Therefore

$$Q = (N \cdot V) \cdot N$$

Now assume that $N$ is a normal to a plane shown as a yellow line. The projection, $P$, of $V$ onto the plane is shown in magenta and is given by $V + G$ where $G$ is the vector shown in green.

Since $G$ and $Q$ have **have the same length** but **point in opposite directions**, $G = -Q$

Therefore the projection of a vector $V$ onto a plane with normal $N$ is given by:

$$P = V + G = V - Q = V - (N \cdot V) \cdot N$$

or

$$\text{vec\_diff}(\text{vec\_scale}(\text{vec\_dot}(N, V), N), V, P);$$

In building your new linear algebra routines, it is desirable to build upon existing ones where possible but extreme levels of nesting of function calls as shown here can **complicate debugging**. (So, in this case, it's probably better to split up the line above into individual parts).

```c
/* project a vector onto a plane */
void vec_project(
    vec_t *n,        /* plane normal     */
    vec_t *v,        /* input vector     */
    vec_t *w)        /* projected vector */
{
}
```
**Reflection**

Basic physics says: The angle of incidence (the angle the incoming ray makes with the normal at the hitpoint) is equal to the angle of reflection.

```c
vec_reflect(
    vec_t  *unitnorm, /* outward surface normal                        */
    vec_t  *unitin,   /* unit vector in incoming direction of the ray  */
    vec_t  *unitout); /* unit vector in outgoing direction ray         */
```

let

\[ U = \text{-} \text{unitin} \quad (\text{notice, this is the reverse of the input vector}) \]

\[ N = \text{unitnorm} \]

then

\[ U + V = 2 \, N \cos(T) \quad \text{where } T \text{ is the angle between } U \text{ and } N \]

\[ \cos(T) = U \cdot N \]

so

\[ U + V = 2 \, N \,(U \cdot N) \]

and

\[ V = 2 \, N \,(U \cdot N) - U \]
**Matrices**

Matrix operations are useful in transforming three-dimensional coordinate systems in ways that make it easier to determine if and where an object is hit by a ray. There are alternative approaches to creating a 3 x 3 matrix.

Probably the most obvious is:

```c
double matrix[3][3];
```

but we will build upon our vector type as follows:

```c
typedef matrix_type
{
    vec_t row[3];
} mat_t;

mtx_t matrix;
```

For example, you can picture a 3 X 3 matrix like the following:

\[
\begin{bmatrix}
1.0 & 1.0 & 0.0 \\
-1.0 & 1.0 & 0.0 \\
0.0 & 0.0 & 1.0
\end{bmatrix}
\]

To set the the element in the 3\textsuperscript{rd} column of the middle row to fifteen, do

```c
matrix.row[1].z = 15.0;
```

which would result in the following matrix:

\[
\begin{bmatrix}
1.0 & 1.0 & 0.0 \\
-1.0 & 1.0 & 15.0 \\
0.0 & 0.0 & 1.0
\end{bmatrix}
\]

Since each row of a matrix is a vector, we can use our vector functions to operate on rows of a matrix. For example, to add the first two rows of a matrix:

```c
vec_t sum;
vec_sum(&matrix.row[0], &matrix.row[1], &sum);
```

Suppose \(V\) and \(W\) are vectors and \(a\) is a scalar value. A function \(F\) that maps three-dimensional space to three-dimensional space is a linear transformation if and only if

\[
F(aV + W) = a F(V) + W \quad \text{for any choice of } a, V, \text{ and } W
\]

Furthermore, any linear transformation may be represented by multiplication of a vector by a matrix.
**Multiplication of a matrix times a vector.**

The product of a 3 x 3 matrix with a 3-d column vector is a 3-d vector. The multiplication rule is as follows:

\[
\text{product}[i] = \text{the dot product of the } i\text{th row of the matrix with the vector.}
\]

\[
\begin{array}{ccc|c|c}
1.0 & 1.0 & 0.0 & 1.0 & 1.0 \\
-1.0 & 1.0 & 0.0 & 0.0 & -1.0 \\
0.0 & 0.0 & 1.0 & -2.0 & -2.0 \\
\end{array}
\]

because (1, 1, 0) dot (1, 0, -2) = 1 + 0 + 0 = 1

because (-1, 1, 0) dot (1, 0, -2) = -1 + 0 + 0 = -1

because (0, 0, 1) dot (1, 0, -2) = 0 + 0 + -2 = -2

The `vec_xform()` function should multiply a vector by a matrix.

```c
void vec_xform(
    mtx_t *m,     /* input matrix             */
    vec_t *v1,    /* vector to be transformed */
    vec_t *v2)    /* output vector            */
{
    vec_t v3;  /* avoid aliasing problems */

    /* Perform the transform using work for output */

    /* Copy work back to v2 */
    vec_copy(&v3, v2);
}
```
The transpose of a matrix:

The transpose of a three by three matrix is also three by the matrix. Its elements are given by a simple rule:

\[ \text{transpose}[i][j] = \text{original}[j][i] \]

\[
\begin{bmatrix}
1.0 & 3.0 & 2.0 \\
1.0 & 2.0 & -2.0 \\
-2.0 & 0.0 & 1.0
\end{bmatrix}
\]

\[ T \]

\[
\begin{bmatrix}
1.0 & 1.0 & -2.0 \\
3.0 & 2.0 & 0.0 \\
2.0 & -2.0 & 1.0
\end{bmatrix}
\]

Notes:
The diagonal elements of a matrix and its transpose are identical. Off diagonal elements are interchanged in a symmetrical way.

The transpose of a matrix is, in general, not the same as the inverse of a matrix.

```c
void mtx_transpose(
    mtx_t  *m1,    /* Input matrix            */
    mtx_t  *m2)    /* Output transpose        */
{
    mtx_t  m3;  /* Avoid aliasing problems */

    m3.row[0].x = m1->row[0].x;
    m3.row[0].y = m1->row[1].x;
    m3.row[0].z = m1->row[2].x;

    // etc....

    // copy m3 back to m2..
}
```
Rotation matrices

Rotation matrices are used to rotate coordinate systems in 3-space. They have some special properties:

- The three rows are mutually orthogonal unit vectors. That is, the dot product of any pair of rows is 0.
- The three columns are also mutually orthogonal unit vectors.
- The inverse of a rotation matrix is its transpose.
- The 1st row of a rotation matrix is a vector which will be mapped to \([1, 0, 0]\) under the rotation. The 2nd row is a vector which will be mapped to \([0, 1, 0]\) and the third row is a vector that will be mapped to \([0, 0, 1]\).
- This example shows that the middle row is mapped to \((0, 1, 0)\)

```
| r0,0  r0,1  r0,2 | | r1,0 | = 0 |
| r1,0  r1,1  r1,2 | | r1,1 | = 1 |
| r2,0  r2,1  r2,2 | | r1,2 | = 0 |
```

Constructing rotation matrices

Suppose \(V\) and \(W\) are orthogonal unit length vectors. Suppose we want to create a rotation matrix \(M\) that will rotate \(V\) into the X-axis and \(W\) into the Z-axis.

```c
vec_t  V;
vec_t  W;
mtx_t  M;

vec_copy(&V, &M.row[0]);   // V will become the X-axis
vec_copy(&W, &M.row[2]);   // W will become the Z-axis

/* The middle row \(Y = Z \times X\) */
vec_cross(&M.row[2], &M.row[0], &M.row[1]);
```
Another example

Suppose V and W are not necessarily orthogonal unit length vectors. Suppose we want to create a rotation matrix $M$ that will rotate W into the Z-axis and force V to lie in the positive Y, Z (X = 0) plane.

```c
vec_t w = {1.0, 0.0, 1.0};
vec_t v = {1.0, 1.0, 1.0};
vec_t v3;
vec_t v4;
mtx_t m1;
mtx_t m2;
```

All rows of rotation matrices must be unit vectors!

```c
vec_unit(&v, &v);
vec_unit(&w, &w);

vec_print(stderr, "v ", &v);
vec_print(stderr, "w ", &w);
```

If we want W to end up on the +Z access and V to lie in the X = 0 plane, then a vector perpendicular to both W and V must end up pointing along the +X axis. Therefore, the vector $V \times W$ must be what gets mapped to the X axis. It is important to note that the cross product is not length preserving in general! Therefore we must renormalize the 1st row of the matrix.

```c
vec_cross(&v, &w, &m1.row[0]);
vec_unit(&m1.row[0], &m1.row[0]);
```

Since W is to be mapped to the positive z-axis we just copy it to the bottom row of the matrix.

```c
vec_copy(&w, &m1.row[2]);
```

Since the missing middle row must be orthogonal to the other two rows it may be computed via the cross product. The order here is important! $Z \times X = Y$ but $X \times Z = -Y$.

```c
vec_cross(&m1.row[2], &m1.row[0], &m1.row[1]);
```
The matrix is now complete so we print it out.

```c
vec_print(stderr, "r0 ", &m1.row[0]);
vec_print(stderr, "r1 ", &m1.row[1]);
vec_print(stderr, "r2 ", &m1.row[2]);
```

We now apply the matrix to V and W

```c
vec_xform(&m1, &v, &v3);
vec_xform(&m1, &w, &v4);
```

and then print the transformed vectors.

```c
vec_print(stderr, "v3 ", &v3);
vec_print(stderr, "v4 ", &v4);
```

The inverse of a rotation is its transpose.

```c
mtx_transpose(&m1, &m2);
```

So if we apply the inverse to v3 and v4, we should get back the original V and W

```c
vec_xform(&m2, &v3, &v3);
vec_xform(&m2, &v4, &v4);

vec_print(stderr, "v3 ", &v3);
vec_print(stderr, "v4 ", &v4);
```
Normalized V and W vectors

\[
\begin{align*}
v & = 0.577 \quad 0.577 \quad 0.577 \\
w & = 0.707 \quad 0.000 \quad 0.707
\end{align*}
\]

The rotation matrix

\[
\begin{align*}
r_0 & = 0.707 \quad 0.000 \quad -0.707 \\
r_1 & = -0.000 \quad 1.000 \quad 0.000 \\
r_2 & = 0.707 \quad 0.000 \quad 0.707
\end{align*}
\]

Transformed V and W. Note that the transformed V lies in the positive Y-Z plane (its x coordinate is 0) and the transformed W is the positive Z axis.

\[
\begin{align*}
v_3 & = 0.000 \quad 0.577 \quad 0.816 \\
v_4 & = 0.000 \quad 0.000 \quad 1.000
\end{align*}
\]

After applying the inverse transformation, the vectors are transformed back to their original values shown at the top of this page.

\[
\begin{align*}
v_3 & = 0.577 \quad 0.577 \quad 0.577 \\
v_4 & = 0.707 \quad 0.000 \quad 0.707
\end{align*}
\]
Parsing the input file – another approach

Parsing is a process in which an input file containing “sentences” written in some language is:

- read in from a file
- tokenized
- analyzed

The semantics of the language determine the actions that are taken during the analysis. Some languages (e.g. the C programming language) are quite complex and some formal mechanisms are needed to process them. Our input language is simple enough that informal ad hoc methods suffice.

A token is a “word” in the language. In this input:

```
camera cam1
{
    pixeldim 800 600
    worlddim 8 6
    viewpoint 4 4 4
}
```

`camera, cam1, {, pixeldim, 800, 600, etc` are tokens. The individual letters making up the words and the digits making up the numbers are not. If (and only if) the language is structured rigidly enough that the position in a sentence in which string values and numeric values can be known in advance, then `fscanf()` can be used as a combination reader/tokenizer.

- `%s` token is a string of 1 or more characters
- `%d` token is an integer value
- `%lf` token is a double precision value

This will be the case for the raytracer.
Model description language

An example sentence in the model description language is:

camera cam1
{
    pixeldim 800 600
    worlddim 8 6
    viewpoint 4 4 4
}

- Each "sentence" in our language begins with an an entity-type identifier.
- Our entity-types will include camera, material, light, spotlight, plane, sphere, etc.
- The entity-type is followed by user defined and arbitrary entity-name.
- entity-types are analogous to data types in C (int, float, double)
- entity-names are analogous to variable names in C (max, min, pixel)
- The entity-name is followed by a collection of entity-attributes enclosed in { }.
- Each entity-attribute consists of an attribute-type specifier followed by attribute-values.

- The entity-types and attribute-types are predefined keywords and will always be spelled as shown.
- The attribute-values will always follow the attribute type.
- The number of values of a particular attribute will never vary.
- The entity-attributes of any entity may appear in any order.
Attribute values map to the data structure associated with a particular entity-type in an obvious way:

camera cam1
{
  pixeldim 800 600
  worlddim 8 6
  viewpoint 4 4 6
}

The attribute values map to the `camera_t` structure in a reasonably obvious way:

```c
typedef struct camera_type
{
  int    cookie;        /* ID's this as a camera         */
  char   name[NAME_LEN]; /* User selected camera name     */
  int    pixel_dim[2];  /* Projection screen size in pix */
  double world_dim[2];  /* Screen size in world coords */
  vec_t  view_point;    /* Viewpt Loc in world coords */
  irgb_t *image;        /* Build image here              */
}  camera_t;
```

**Note:** There is no direct correspondence between an *attribute name* in the model description language and the *variable name* used to hold the *attribute values*. 
In summary, our model description language looks informally like:

```plaintext
entity-type entity-name
{  
  attribute-type attribute-value(s)  
  attribute-type attribute-value(s)  
  :  
}
entity-type entity-name
{  
  attribute-type attribute-value(s)  
  :  
}
:  
end-of-file
```

That is,

- the attribute list of each entity definition is *terminated by the \} token* and
- the complete model definition is terminated by end-of-file.

Use of the model description in the ray tracer.

- There is a one-to-one correspondence between sentences in the model description language and instances of structures (or structure hierarchies) in the executing raytracer.
- For each sentence read, a new structure must be dynamically created and attributes of the structure filled in using the attribute-name / attribute value pairs supplied in the model description.
- We call the process of digesting the model description and producing the structure instances parsing.
A hierarchical parser

We will use a two level hierarchy for parsing the model description language:

- The top level parser will be responsible for consuming entity-type names
- The entity-type name will identify the type of object (camera, material, light, etc) that must be constructed.
- It will then invoke a constructor for the object to be created.
- The constructor will allocation the required structure read the attribute values into it.

Constructors and parsing

In object oriented languages, each object type has an associated constructor function that is called each time a new instance of the object type is created. Therefore, if a raytracing model contains two planes, the plane constructor will be invoked twice.

*Each entity-type will provide a constructor function that will know about the attributes that apply to the entity will and be responsible for parsing them.*

Entity constructors should use the assert( ) mechanism to abort the program whenever:

- an unknown attribute type is encountered
- the proper number of attribute values cannot be read in
- required attributes are missing

We will constrain, to some degree, the order in which attribute-types may appear.

A general attribute parser

After writing parsers for the camera, material, and plane, the typical programmer will find repeatedly rewriting (almost) the same code tiresome and tedious and will seek a better way. It is not bad that the ad hoc approach was used initially. The very use of the ad hoc helps the programmer see common aspects of the problem and develop a more general solution. As usual we try to make our solution data driven to the extent possible.

To build a general parser we will build upon the capability of the table_lookup mechanism but replace the old table of attribute names with new tables that contain not only the attribute name but also sufficient information to allow the general parser to load the values. Specifically, for each attribute, we need the following information:

- How many values must be loaded (e.g. 2 for pixeldim, 3 for viewpoint)
- How many bytes of storage does each value occupy (4 for pixeldim, 8 for viewpoint).
- What format string should be used to read a value (%d for pixeldim, %lf for viewpoint)
- Where should the first value be stored?

Here we make use of the fact that we know adjacent array elements and members of a structure such as a vec_t are stored in adjacent memory locations. For the vec_t if the location of the x component is memory address a, and then the y component is at location a+8, and the z at location a+16.
The *struct pparm_type*

Therefore, each entity will employ a table in which each attribute is represented by a structure of the following type.

```c
/* the parse parameter structure */
typedef struct pparm_type
{
    char *attrname;      /* Attribute name */
    int numvals;        /* Number of attribute values */
    int valsize;        /* Size of attribute in bytes */
    char *fmtstr;        /* Format string to use */
    void *loc;           /* Where to store 1st attr value */
} pparm_t;
```

**Building tables of attribute descriptors**

For the *camera* entity. We build the structure as follows:

```c
pparm_t camera_parse[] =
{
    {"pixeldim", 2, sizeof(int), "%d", 0},
    {"worlddim", 2, sizeof(double), "%lf", 0},
    {"viewpoint", 3, sizeof(double), "%lf", 0}
};
#define NUM_ATTRS (sizeof(camera_parse) / sizeof(pparm_t))
```

Items to note:
- *camera_parse* is an array of three elements
- each element is a structure of type *pparm_t*
- initializers should be enclosed in {}
- the location where the first attribute value should be stored will live in the *camera_t* structure that is eventually allocated with *malloc()* . These values can't be set until after the *camera_t* is *malloc'd*.

For the other entities such as the *material* entity, the structure is analogous.

```c
pparm_t mat_parse[] =
{
    {"ambient", 3, sizeof(double), "%lf", 0},
    {"diffuse", 3, sizeof(double), "%lf", 0},
    :}
#define NUM_ATTRS (sizeof(mat_parse) / sizeof(pparm_t))
```
The interface to the general attribute parser

This version of `camera_init()` demonstrates how the use of the general parser can reduce your workload.

```c
void camera_init(FILE *in, model_t *model, int attrmax)
{
    /* malloc the camera structure */
camera_t *cam = malloc(sizeof(camera_t));
    assert(cam != NULL);
cam->cookie = CAM_COOKIE;

    /* Read camera name and { */
    fscanf(in, "%s", cam->name);
    fscanf(in, "%s", buf);
    assert(buf[0] == '{');

    /* Store locations where attribute data should be read */
camera_parse[0].loc = &cam->pixel_dim;
camera_parse[1].loc = &cam->world_dim;
camera_parse[2].loc = &cam->view_point;

    /* Invoke the parser */
    mask = parser(in, camera_parse, NUM_ATTRS, 0);

    /* verify required attributes read */
    assert(mask == 7);

    /* remember address of camera structure */
    model->cam = cam;
}
```
/* Generalized attribute parser */
/* It returns a bit mask in which each possible attribute */
/* is represented by a bit on exit the attributes that    */
/* have been found will have their bit = 1                */

int parser(
    FILE *in,
    pparm_t *pct,         /* parser control table                */
    int      numattrs,    /* number of legal attributes          */
    int      attrmax)     /* Quit after this many attrs if not 0 */
{]

    char attrname[NAME_LEN];
    int attrcount = 0;   /* number of attribs loaded */
    int mask = 0;        /* loaded attrib bit mask */
    int ndx;             /* ndx of this attrib in pct */

    /* One trip is made through this loop for every attribute */
    /* processed... Exit from the loop is triggered by '}' */
    /* or if the maximum number of attributes is set, when */
    /* the maximum number have been processed */

    fscanf(in, \"%s\", attrname);
    while (strlen(attrname) && attrname[0] != '}') {
        /* Process one attribute */
        ndx = parser_load_attr(in, pct, numattrs, attrname);
        mask |= 1 << ndx;
        attrcount++;

        /* See if its quitting time -- */
        if ((attrmax) && (attrcount == attrmax))
            break;
        *attrname = 0;
        fscanf(in, \"%s\", attrname);
    }

    if (attrmax != attrcount)
        assert(attrname[0] == ')');
    return(mask);
}
Loading the values of a single attribute

```c
int parser_load_attr(  
FILE *in,  
pparm_t *pct, /* parser control table */  
int numattrs, /* number of legal attributes */  
char *attrname) /* attribute name */  
{
  pparm_t *pce; /* Entry corresp to this attribute */  
  int count = 0;
  unsigned char *loc; /* where to store value.. have to */  
  double *work; /* use unsigned char for pointer */  
  int ndx; /* arithmetic to work correctly */  
  int i;

  /* table_lookupp is an updated version of table_lookup that */  
  /* takes parse control table pointer as input. */  
  ndx = table_lookupp(pct, numattrs, attrname);
  assert(ndx >= 0);

  /* Point to the proper entry in the table */  
  pce = pct + ndx; // or pce = &pct[ndx];

  /* pce->loc points to where the first value must go */  
  loc = (unsigned char *) pce->loc;

  /* Attributes may have different numbers of attribute values */  
  /* for example the viewpoint has three but the pixeldim only */  
  /* has 2 values. Each iteration consumes one value */  
  for (i = 0; i < pce->numvals; i++) {
    count += fscanf(in, pce->fmtstr, loc);
    // work = (double *)loc;
    // fprintf(stderr, "%s %lf \n", pce->attrname, *work);
    loc += pce->valsize; // point to next spot
  }
  assert(count == pce->numvals);
  return(ndx);
}
```

Exercise: Design a generic table_lookup function